2006-09

Design, implementation and testing of a common data model supporting autonomous vehicle compatibility and interoperability

Davis, Duane T.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

DESIGN, IMPLEMENTATION AND TESTING OF A COMMON DATA MODEL SUPPORTING AUTONOMOUS VEHICLE COMPATIBILITY AND INTEROPERABILITY

by

Duane T. Davis

September 2006

Dissertation Supervisor: Don Brutzman

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2006	**3. REPORT TYPE AND DATES COVERED** Doctoral Dissertation	
4. TITLE AND SUBTITLE: Design, Implementation and Testing of a Common Data Model Supporting Autonomous Vehicle Compatibility and Interoperability			**5. FUNDING NUMBERS**
6. AUTHOR: Duane T. Davis			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT ***(maximum 200 words)***

Current autonomous vehicle interoperability is limited by vehicle-specific data formats and support systems. Until a standardized approach to autonomous vehicle command and control is adopted, true interoperability will remain elusive. This work explores the applicability of a data model supporting arbitrary vehicles using the Extensible Markup Language (XML). An exemplar, the Autonomous Vehicle Command Language (AVCL), encapsulates behavior-scripted mission definition, goal-based mission definition, inter-vehicle communication, and mission results.

Broad applicability is obtained through the development of a behavior set capturing arbitrary vehicle activities, and automated conversion of AVCL to and from vehicle-specific formats. The former uses task-level behaviors suitable for mission scripting and goal decomposition. Translations use the Extensible Stylesheet Language for Transformation, XML data binding, context-free language parsing, and artificial intelligence machine learning and search techniques. Translation capability is demonstrated through mappings of AVCL to and from multiple vehicle-specific formats.

A final demonstration of the power of a common autonomous vehicle data model is provided by the implementation of a hybrid control architecture. The model's vehicle-independence and the ability to generate vehicle-specific data are leveraged in the design of an architecture that provides increased autonomy by augmenting a vehicle's existing controller. The utility of this architecture is demonstrated through implementation on the Naval Postgraduate School's ARIES Unmanned Underwater Vehicle.

14. SUBJECT TERMS AUV, UUV, USV, UAV, robotics, autonomy, control architecture, hybrid control, autonomous vehicle behaviors, state-based control, data model, ontology, XML, XSLT, XML data binding, context-free grammar, data translation			**15. NUMBER OF PAGES** 359
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DESIGN, IMPLEMENTATION AND TESTING OF A COMMON DATA MODEL SUPPORTING AUTONOMOUS VEHICLE COMPATIBILITY AND INTEROPERABILITY

Duane T. Davis
Commander, United States Navy
B.S., Virginia Polytechnic Institute and State University, 1989
M.S., Naval Postgraduate School, 1996

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: _____
Duane T. Davis

Approved by:

_____
Don Brutzman
Associate Professor of Applied Science
Dissertation Supervisor and
Dissertation Committee Chair

_____ _____
Neil Rowe, Professor of Computer Science
Robert McGhee, Professor of Computer Science

_____ _____
Christian Darken, Associate Professor of Computer Science
Anthony Healey, Distinguished Professor of Mechanical Engineering

Approved by: _____
Peter Denning, Chair, Department of Computer Science

Approved by: _____
Julie Filizetti, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Current autonomous vehicle interoperability is limited by vehicle-specific data formats and support systems. Until a standardized approach to autonomous vehicle command and control is adopted, true interoperability will remain elusive. This work explores the applicability of a data model supporting arbitrary vehicles using the Extensible Markup Language (XML). An exemplar, the Autonomous Vehicle Command Language (AVCL), encapsulates behavior-scripted mission definition, goal-based mission definition, inter-vehicle communication, and mission results.

Broad applicability is obtained through the development of a behavior set capturing arbitrary vehicle activities, and automated conversion of AVCL to and from vehicle-specific formats. The former uses task-level behaviors suitable for mission scripting and goal decomposition. Translations use the Extensible Stylesheet Language for Transformation, XML data binding, context-free language parsing, and artificial intelligence machine learning and search techniques. Translation capability is demonstrated through mappings of AVCL to and from multiple vehicle-specific formats.

A final demonstration of the power of a common autonomous vehicle data model is provided by the implementation of a hybrid control architecture. The model's vehicle-independence and the ability to generate vehicle-specific data are leveraged in the design of an architecture that provides increased autonomy by augmenting a vehicle's existing controller. The utility of this architecture is demonstrated through implementation on the Naval Postgraduate School ARIES unmanned underwater vehicle.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENT

It would be impossible to list all of the people upon whom I relied over the course of this research. I would, however, like to take this opportunity to acknowledge a few of the people whose support and encouragement proved invaluable:

Don Brutzman, for his time, insight and friendship. Without his assistance this work would not have been possible. He was relentless, and for that I am eternally grateful.

Robert McGhee for his continuing mentorship and inspiration. Without his encouragement I would never have even begun this process. It is an honor to have his signature on this document.

Anthony Healey for his ideas and insights and for allowing me to test my ideas on his vehicle. His seemingly unlimited knowledge, experience, and passion for the science are truly inspirational.

Neil Rowe for his across-the-board support from start to finish. Whether helping with qualification exam preparation, helping me to view class material in the context of this research, or offering ideas for exploration, he was always available and his input was always on the mark.

Chris Darken for always making himself available to critique ideas and provide advice. His ideas formed the basis of a number of avenues of exploration described here.

My fellow Ph.D. students. Whether it was at happy hour, one of our semi-regular breakfasts, or idle chatter in the office, they made the experience a lot more pleasant than it would have been without them.

And finally, but most importantly, my lovely wife, Cecelia. Of all of the things with which I have been blessed during my time in Monterey, I am most grateful for her.

I would also like to acknowledge the financial support of the Navy Modeling and Simulation Office, Naval Research Laboratory, Stennis Space Center, Naval Undersea Warfare Center, Newport, Naval Facilities Engineering Service Center, and the Defense Science Technology Office, Singapore.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

$-Calculus	Cost Calculus
A*	A-Star Search
ADEPT	All-Domain Execution and Planning Technology
API	Application Programmer's Interface
ARIES	Acoustic Radio Interactive Exploratory Server
AUV	Autonomous Underwater Vehicle
AUVW	Autonomous and Unmanned Vehicle Workbench
AVCL	Autonomous Vehicle Command Language
C2L	Compact Control Language
CCL	Common Control Language
CNO	Chief of Naval Operations
CoDA	Cooperative Distributed Autonomous Oceanographic Sampling Network
D*	Focused Dynamic A* Search
DAML+OIL	DARPA Agent Modeling Language + Ontology Inference Layer
DAMN	Distributed Architecture for Mobile Navigation
DARPA	Defense Advanced Project Agency
DIS	Distributed Interactive Simulation protocol
DOD	Department of Defense
DOM	Document Object Model
DON	Department of the Navy
DTD	Document Type Definition
DVD	Digital Versatile Disc
ERBM	Extended Rational Behavior Model
ESRI	Environmental Systems Research Institute
FIPA	Foundation for Physical Agents
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
ICAO	International Civil Aviation Organization
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IMO	International Maritime Organization
ISO	International Organization for Standardization
IST	Information Society Technical Programme
ITU	International Telecommunication Union
JAUS	Joint Architecture for Unmanned Systems
JAXB	Java Architecture for XML Binding
JC3IEDM	Joint Consultation, Command and Control Information Exchange Data Model
JRP	Joint Robotics Program
kΩ	K-Omega Optimization
MIP	Multilateral Interoperability Programme
NAS	National Academy of Sciences

NAVO	Naval Oceanographic Office
NPS	Naval Postgraduate School
NP-hard	Nondeterministic-polynomial-time-hard computational complexity
OGC	Open Geospatial Consortium
OIAG	Open Inventor Architecture Group
ONR	Office of Naval Research
OMG	Object Management Group
OWL	Web Ontology Language
RBM	Rational Behavior Model
REMUS	Remote Environmental Sensing UnitS
SAX	Simple API for XML Parsing
SGML	Standard Generalized Markup Language
T*	Time-Bounded A* Search
UAV	Unmanned Air Vehicle
UGV	Unmanned Ground Vehicle
USV	Unmanned Surface Vehicle
UUV	Unmanned Underwater Vehicle
W3C	World Wide Web Consortium
X3D	Extensible Three-Dimensional Graphics
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XPath	XML Path Language
XSBC	XML Schema-Based Binary Compression
XSLT	Extensible Stylesheet Language for Transformations

I. INTRODUCTION, MOTIVATION, AND OBJECTIVES

A. DISSERTATION STATEMENT

Recent events have highlighted numerous military and civilian applications for which autonomous vehicles might prove useful. Further, a number of these applications can benefit from the use of multiple, possibly dissimilar autonomous vehicles operating in a cooperative, or at least complementary, manner. Unfortunately, currently available autonomous vehicles are, by and large, designed to operate as independent entities making coordinated multi-vehicle missions unfeasible from a practical standpoint. Additionally, vehicle-specific data formats and mission planning systems make planning complementary multi-vehicle missions (i.e., missions in which multiple vehicles operate independently to accomplish common goals without direct interaction or cooperation) problematic.

Significant recent research has investigated methodologies and protocols to foster interoperability among autonomous vehicles, however, the preponderance of this research has assumed that the vehicles involved are inherently compatible. That is, either the multi-vehicle system consists solely of one type of vehicle, or all vehicles use the same language for mission specification and inter-vehicle communication. Unfortunately, assumed compatibility is unrealistic given current inventories of legacy vehicles and the parallel development of vehicles by various commercial, academic, and government entities.

The preceding observations call into question the inherent suitability of both current and developmental vehicles for multi-vehicle operations: currently available autonomous vehicles do not directly support coordinated operations, and forthcoming vehicles will depend on vehicle-specific data formats and protocols to provide compatibility with similarly programmed vehicles. It is this implicit shortcoming in the capabilities of both current and envisioned autonomous vehicles that this research addresses.

The main thrust of this work is to demonstrate the use of a common data model or ontology to foster a level of compatibility among autonomous vehicles regardless of

inherent vehicle differences. Ultimately, the compatibility provided by the common data model will foster inter-vehicle coordination and cooperative operations. Discussed in some detail in Chapter IV, the term ontology refers to a formal description of a vocabulary, including word meanings, assumptions and relationships, that can be used to describe and represent an area of knowledge (Daconta, et al., 03), in this case autonomous vehicle operations. An exemplar has been defined and implemented using the Extensible Markup Language (XML) schema (W3C 04-2)(W3C 04-3) and utilized as a framework for autonomous vehicle tasking and inter-vehicle communications between dissimilar vehicles. Further, the ability to automatically convert between this model-constrained format and vehicle-specific formats as required is demonstrated.

Additionally, this research uses the exemplar data model to improve upon existing autonomous vehicle control paradigms by incorporating multiple levels of a layered control architecture in the same data model. Specifically, an extension to the Rational Behavior Model (RBM) (Byrnes, 93), a hybrid control architecture, is implemented that connects high-level (Strategic) control based on a declarative goal-based mission specification in the form of a finite state machine with mid-level (Tactical) control using sequentially initiated task-level behaviors.

B. MOTIVATION AND OVERVIEW

1. The Multiple Autonomous Vehicle Inter-Operability Requirement

The military services have recognized the distinct advantages provided by the utilization of unmanned vehicles for a variety of missions that are considered too "dirty, dull, or dangerous" for humans (JRP, 04). Unmanned Underwater Vehicles (UUV) were recently used in mine countermeasure and search and rescue operations in support of Operation Iraqi Freedom, crash and salvage operations in support of space shuttle *Discovery* recovery operations, and berthing surveys in support of port security (Simmons, 04). While none of these operations involved multi-vehicle operations, they have provided a glimpse of the reliance on UUVs in future naval operations.

The U.S. Navy's Unmanned Undersea Vehicle Master Plan has identified nine potential signature capabilities of particular interest including intelligence, surveillance and reconnaissance, mine countermeasures, communication / navigation network node, oceanography, and anti-submarine warfare (DON, 04). Capability evolution in each of

these areas is anticipated to involve increasing autonomy as well as increasing numbers of vehicles operating together. Two exemplar anticipated evolutions in the signature capabilities involving coordinated multiple-UUV capabilities are mobile undersea networks (communications / navigation aid) and Shallow Water Autonomous Reconnaissance Modules (DON, 04)(St. Peter and LaPoint, 04). Autonomy and communication are considered key technology requirements for all four signature capabilities, while "multiple vehicles working together" is identified as having associated risks that must be addressed in order to achieve the target capabilities (DON, 04).

A broader analysis of the potential application of autonomous vehicles in a naval environment can be found in the National Academy of Sciences (NAS) Report on Unmanned Vehicles in Support of Naval Operations (NAS, 05) which identifies a number of potential military missions autonomous vehicles. In addition to reiterating the UUV missions described in the Navy's UUV Master Plan, this report discusses the potential use of unmanned air (UAV), ground (UGV), and surface (USV) vehicles for intelligence, surveillance and reconnaissance, suppression of enemy air defense, amphibious landing zone reconnaissance, concealment area (e.g., caves or building interiors) investigation, ordnance disposal and mine clearance, logistics, and shore bombardment fire control observation. In addition to a detailed discussion of potential uses, this report rigorously analyzes the applicability of current technologies, identifies cultural, research, and acquisition shortcomings that inhibit the development of mission-suitable vehicles, and proposes roadmaps to achieve required capabilities.

A product similar to the Navy's Unmanned Undersea Vehicle Master Plan is the Department of Defense Joint Robotics Program Fiscal Year 2004 Master Plan which focuses on UGVs. This document outlines the near and mid-term goals for UGVs and defines development priorities to achieve them. While not directly addressing multi-vehicle operations, it does address a number of target missions for which coordinated multi-vehicle operations might be advantageous including reconnaissance, surveillance and target acquisition, detection and neutralization of minefields, force protection, and contaminated area operations (JRP, 04). Excerpted from this work, Figure 1.1 illustrates the projected transition from remotely commanded to fully autonomous operations that was identified by (DON, 04) as a key technology. Among the primary near-term goals

identified is the development of a common architecture for unmanned vehicles—the Joint Architecture for Unmanned Systems (JAUS) (JRP, 04), which is covered in more detail in Chapter II. For now it is sufficient to note that JAUS defines components that can be combined to design arbitrary unmanned or autonomous vehicles as well as a message set for communication between individual components or vehicles. A significant aspect of JAUS that it is platform independent so long as the platform is "JAUS compliant."

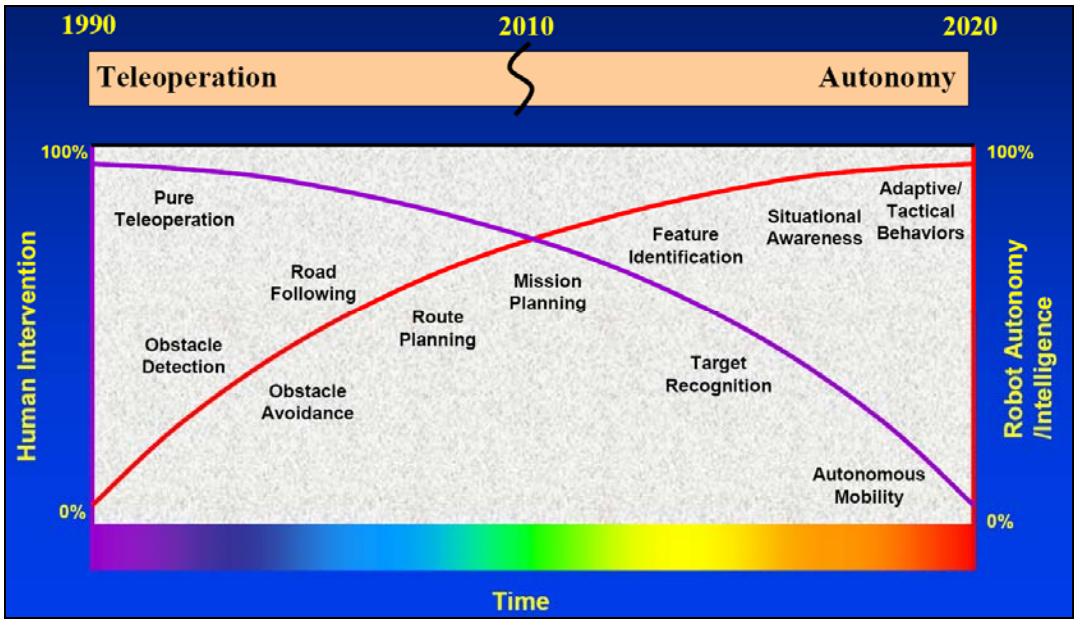


Figure 1.1. Projected Evolution of the Level of Autonomy in Unmanned Vehicles from 1990 to 2020 (From: JRP, 04)

UAV utilization has been well documented in recent military operations, but the vehicles concerned were primarily remotely operated vice autonomous. Additionally, the military services have not, as yet, developed a master plan for UAV development or procurement, but tactical UAV utilization has been doctrinized to some extent in service-specific operational guides such as (CNO, 04). Unfortunately, as UAV tasking increases, the limitations of remotely operated vehicles due to operator overload are being reached. The obvious solution is to use increased autonomy to enable individual operators to monitor and control more vehicles simultaneously (McLoud and Wu, 04). A predictable evolution of increased individual UAV autonomy as an enabler of multi-vehicle operations is increased multi-vehicle autonomy as evidenced by recent research in the area of cooperative planning for UAVs (Ahner, 04)(Neushul, 03).

To date, the use of unmanned and autonomous vehicles for nonmilitary applications, whether as single units or coordinating groups, has been somewhat limited and does not approach the extent of utilization in military settings, at least in part due to cost-versus-gain issues. However many of the military applications for unmanned and autonomous vehicles have fairly obvious non-military parallels. For instance many aspects of surveillance and interdiction are applicable to homeland defense and border security while the search and detect aspects of mine countermeasures closely parallel those of maritime search and rescue or crash and salvage operations. As an example, the National Oceanic and Atmospheric Administration has proposed autonomous underwater vehicles for coastal survey, fisheries management, ocean exploration, and physical oceanographic research (Manley, 04).

2. A Common Data Model as a Coordinated Operations Enabler

Notwithstanding that coordinated multi-vehicle operations are now generally acknowledged as being desirable, the preponderance of research to date assumes that the vehicles involved are inherently compatible. That is, either the multi-vehicle system consists solely of one type of vehicle, or all vehicles use the same language for mission specification and inter-vehicle communication. This is unrealistic given current inventories of diverse legacy vehicles and the unsynchronized, parallel development of vehicles by various commercial, academic, and government entities, but a common data model can go a long way in filling this interoperability shortcoming.

Fortunately, arbitrary vehicles of a given type have significant functional similarities regardless of mission specification methodology and available vehicle-specific commands. For instance, while individual autonomous ground vehicles may have different means and speeds of locomotion, spatial positioning and orientation for any ground vehicle can be ordered as a waypoint (geographic position and possibly heading). Speed (as a percentage of max speed) and path can also be part of the order, or can be left to the individual vehicle (which presumably has embedded local path planning and obstacle avoidance). This inherent similarity in the capabilities of all vehicles of a given type (and even those of different types in many cases) allows for the definition of a set of task-level or script commands that can be used individually or as composite commands to capture the semantics of arbitrary vehicle-specific commands. In short, for

any mission defined in a vehicle-specific language, an equivalent mission can be defined using task-level behaviors defined by a more general data model.

The ability to define vehicle-independent missions that are equivalent to arbitrary vehicle-specific missions is only part of the problem—in and of itself, a common data model is not the goal. Rather, it needs to serve as a bridge between existing and future vehicle-specific languages as well as their human operators. That is, a mechanism must exist to automatically convert between vehicle-specific data formats and the common data model. This research demonstrates the use of XML technologies such as the Extensible Stylesheet Language for Transformations (XSLT) and XML data binding to convert exemplar data-model-compliant documents into arbitrary vehicle-specific formats. Similarly, context-free grammar definitions for vehicle-specific data formats are used as a linchpin of automated translation of vehicle-specific data into data-model-compliant XML. Discussed in detail in Chapter V, this procedure provides a suitable methodology for automated conversion between arbitrary vehicle-specific data formats by leveraging the generality of a vehicle-independent data model with XSLT, XML data binding and context-free-grammar-based parsing of non-XML data formats.

Intervehicle communication is essential to effective coordinated operations. As with tasking, individual vehicles have various communications capabilities, protocols and message sets that must be reconciled if they are to interoperate. A methodology similar to that used to convert missions from one vehicle-specific language to another is used to convert vehicle-specific communications as required. An exemplar architecture for unmanned system messaging is provided by JAUS which incorporates an explicitly defined message set and format (JAUS 04-4). Although JAUS in its present form is more suitable for remotely operated vehicles than autonomous vehicles, similarities exist between a number of JAUS messages and communications aspects of the exemplar data model developed in this research. As with mission specification, XSLT, XML data binding, and context-free grammars are used to convert between data-model-compliant XML and vehicle-specific message formats (including JAUS when appropriate). Additionally, XML Schema-Based Binary Compression (XSBC) is used to facilitate the transmission of admittedly bulky XML messages over noisy and bandwidth-limited communications paths (XSBC is covered in Chapter III).

Automated conversion of vehicle-specific data from one robot-tasking format to another is a powerful capability that can significantly enhance interoperability between dissimilar vehicles, but it does not directly address increased autonomy. A vehicle-independent data model, however, can be designed in such a way as to support increased autonomy. The mission scripting task-level behaviors are by nature well suited to many artificial intelligence planning algorithms. Each command has preconditions that must be true in order for it to execute as well as postconditions that are expected to be true upon successful completion. A higher-level, goal-based mission specification vocabulary can be designed to take advantage of this observation. Goal-based missions are defined declaratively using this vocabulary, and planning algorithms generate sequential plans that achieve the declarative goals using task-level behaviors. Once generated, the task-level scripts are converted into vehicle-specific missions for execution. This capability is demonstrated off line and in real time as an implicit function of a multi-layer vehicle-independent control architecture.

3. Data Model Use for Cross-Application Data Sharing

Knowledge representation has long been a topic of interest in the field of artificial intelligence but the design of data models to facilitate dissimilar system interoperability is less explored. The evolution of the Internet, however, has led to an increased desire to share information between applications and systems and also to make data more easily accessible. In the database domain this trend has led to the development of protocols and application programmer's interfaces (API) along the lines of the Open DataBase Connectivity standard. On a broader scale XML and associated technologies have emerged as a mechanism for facilitating structured data exchange among highly disparate systems. In particular, the use of XSLT has become the method of choice for mappings between the various data models of dissimilar systems and applications (Kay, 03). Further, the notion of the Semantic Web is a primary driver behind current research into ontologies and conceptual data models. Semantic Web approaches often include the goal of enabling applications to automatically locate and utilize available data sources without *a priori* knowledge of their content, format or even their existence (Daconta, et al., 02).

Given the growth of electronic commerce in recent years, it is understandable that much current research in this area focuses on business applications relying on the Internet

(Fensel, 01). Efforts along the lines of the Joint Consultation, Command and Control Information Exchange Data Model (JC3IEDM), however, provide an indication that common data models can effectively facilitate information exchange between systems that are not specifically designed to operate together regardless of whether the exchange takes place over the internet or not. JC3IEDM is a multi-lateral effort to design a data model for the exchange of command and control information between military systems. Although JC3IEDM and its applicability to this research are discussed in more detail in Chapter II, it is worth noting here that JC3IEDM places no requirements whatsoever on the internal data models of individual applications—it is concerned solely with data that is exchanged. That is, individual applications and systems can represent and interpret internal data in any way, but in order to exchange data with another system, it must be JC3IEDM-compliant (MIP, 03-1).

The application of a common data model to the domain of autonomous vehicle command and control as implemented in this research takes a similar approach. With the exception of the data-model-centric multi-layer controller discussed in Chapter VII, individual vehicles are not required to utilize the data model internally. One difference between the approach of this research and that of JC3IEDM is a relaxation of the requirement for model-compliant data transfers so long as the data model is used as the bridge between systems. In other words, the data transmission can be in any format so long as it is converted from the transmitting system's native format into a common data-model-compliant format and eventually into the receiving system's native format. This provides the flexibility to perform the required translations at the most appropriate location to take advantage of vehicle capabilities, deal with communications path limitations, etc.

C. OBJECTIVES

This dissertation addresses the following research questions:

- Can a single data model be used to accurately represent tasking for arbitrary autonomous vehicles? The obvious (but unwieldy) answer is to explicitly incorporate the tasking semantics of all vehicles of interest into the data model. This, however, provides little basis for automatic translation between the model and arbitrary vehicle-specific formats. A more tenable solution takes advantage of the fact that despite differing lexical and semantic tasking formats, at the most basic levels, autonomous

vehicle tasking is implemented through the deliberate execution of one or more elements from a finite set of simple tasks or behaviors. Rephrased, most robots share similar functional capabilities despite major differences in communications and tasking styles. Identifying a suitable set of task-level behaviors is, therefore, a key to the development of a useful common data model for autonomous vehicle tasking. Ultimately, the elements of this set can be used to capture the semantics of arbitrary vehicle-specific commands under the umbrella of a single ontology.

- Can a common ontology or data model be utilized as a bridge between vehicle-specific tasking and communications languages of incompatible vehicles? That is, can a common data model be defined in such a way as to facilitate the automated conversion of arbitrary vehicle-specific data to model-compliant data and vice versa? Resolution of this issue depends on the previous research question—implementation notwithstanding, if arbitrary vehicle-specific data formats cannot be accurately represented in the common data model, conversion (automated or not) between the common data model and vehicle-specific formats will not be possible. Assuming the availability of an appropriate task-level behavior set, translation mechanisms must still be developed. Implementation of the data model using XML provides a partial solution. Through the use of XML utilities such as the XSLT and XML-data binding, data-model-compliant data is translated into arbitrary vehicle-specific formats without difficulty. The requirement to convert non-XML vehicle-specific data formats into the common data model is more problematic. Recognizing, however, that vehicle-specific data formats are in reality context-free languages provides a basis for automated translation (Crangle and Suppes, 94). Formal definition of vehicle-specific data formats using context-free grammars enables the generation of parse trees that are traversed in depth-first order and translated to the common data model using templates.

- Can the data model described in the previous two paragraphs be expanded to capture vehicle-specific data of a more symbolic nature? While the task-level control paradigm is suitable for most currently available vehicles, a number of autonomous vehicle control architectures that support higher-level reasoning have been the subject of recent research (Byrnes, 93)(Rosenblatt, 97)(Ricard and Kolitz, 02)(Stentz, 04). These control architectures are at least theoretically capable of interpreting data of a more symbolic or declarative nature (e.g., tasking as a set of high-level goals as opposed to a sequentially executed script). One implication relating to the use of a common data model raised by this potential for declarative tasking is the required capture of the declarative semantics by the data model—something beyond the capability of the task-level behavior set. It is therefore necessary to explicitly provide for symbolic definition of missions. In the exemplar data model, declarative missions are defined as in the form of a finite state machine. Individual states correspond to individual mission goals (e.g., search a geographic area) and

transitions are executed when the vehicle succeeds (or fails) in the accomplishment of the goal corresponding to the current state.

- A second, implication of declarative mission definition is the requirement for conversion between declarative and task-level mission definitions. For a common autonomous vehicle data model to be compatible with both vehicles utilizing task-level behaviors and those utilizing symbolic mission definitions, mechanisms must be developed for converting between declarative symbolically defined missions and sequential task-level defined missions. The previously discussed XSLT and context-free-grammar-based conversions do not suffice for this purpose. This research demonstrates that translation of symbolic missions to task-level missions can be accomplished through fairly traditional artificial intelligence planning algorithms such as means-ends analysis, GraphPlan and A-Star (A*) search (Luger, 02)(Russell and Norvig, 03). Upward translation of task-level missions to symbolic missions is a more difficult problem because such interpretation requires inference of intent. That is, what goal is a particular sequence of task-level behaviors intended to accomplish? This research explores the use of machine learning techniques such as case-based reasoning and naïve Bayesian analysis to infer reasonable "intent" from task-level scripts.
- A final research question concerns the use of a common data model to support and extend existing autonomous vehicle control architectures. Common characteristics of hierarchical and hybrid control architectures (discussed in Chapter II) are a top-most level utilizing a general or abstract definition of the mission to direct the activities of a lower level through the use of specific vehicle behaviors or primitive objectives. The previously discussed translation of declarative missions to ordered lists of task-level behaviors is similar enough to the requirements of multi-layered control architecture implementation that it raises the following question: can a common data model capable of representing both declarative and task-level missions be utilized as an integral part of a multi-layer control architecture? This possibility is explored by extending the RBM to utilize the common data model symbolic mission definition capability at the Strategic level and the task-level behavior set at the Tactical and Execution levels.

D. DISSERTATION ORGANIZATION

Chapter II of this dissertation comprises an overview of related work. Of specific interest are other languages, data formats, and methodologies currently used for autonomous vehicle command, mission specification, multi-vehicle system definitions, and communications along with the relative strengths and weaknesses of reach. Chapter III provides an overview of XML and applicable XML technologies that are utilized to demonstrate the concepts and methodologies explored in this research. Chapter IV

begins with a discussion of the similarities and differences between ontologies and data models, and concludes with an overview of data-model requirements for the support of this work including the task-level behavior set, declarative mission specification, communications and mission results data. Also included in Chapter IV is a brief discussion of the exemplar data model implemented in support of this research—the Autonomous Vehicle Command Language (AVCL). Chapter V provides a detailed analysis of the approaches for converting between vehicle-specific formats and the vehicle-independent data model. Chapter VI covers the implementation of planning algorithms and other methodologies for conversions between declaratively defined missions to task-level behavior scripts and the inference of appropriate goals corresponding to task-level behavior scripts. Chapter VII deals with the extension of the RBM architecture using the exemplar data model and the implementation of this extended architecture on an existing UUV. Chapter VIII covers simulation and real-world experimentation and results supporting this research. The dissertation concludes with Chapter IX's discussion of conclusions to be drawn from this research and recommendations for future work. Appendices are provide to describe the specific content structure and semantics of the exemplar data model and the mission-planning, rehearsal and simulation application that was developed to support this research.

THIS PAGE INTENTIONALLY LEFT BLANK

II. REVIEW OF RELATED WORK

A. INTRODUCTION

Two areas of current research that are relevant to the development of a common autonomous vehicle ontology are addressed in this chapter. First is a survey of commonly utilized autonomous vehicle control architectures, their relative strengths and weaknesses and the potential means in which this research complements or augments them. The main point to be taken from this portion of the discussion is that a common data model is capable of supporting any of these methods, thus providing a means of facilitating interoperability between autonomous vehicles utilizing different control paradigms.

Following the discussion of general autonomous vehicle control architectures, a brief discussion of the RBM architecture is provided in order to illustrate how a common data model supporting abstract and declarative mission definition along with sequential task-level mission definition can be inherently compatible with a multi-layered architecture.

The final section of this chapter covers research in the area of platform-independent languages, architectures and data models. A number of proposed autonomous vehicle programming languages and architectures potentially falling into this genre such as Robotalk (Phoha and Schmiedekamp, 04), Yampa (Hudak, et al., 03) and Player / Stage (Vaughan, et al., 03) are not covered because they are utilized to program autonomous vehicle controllers symbolically in the same manner that Lisp or Prolog might be used to develop traditional artificial intelligence applications. Of more interest from the standpoint of a common data model are those languages and data formats that are used to define individual missions in a relatively abstract and straightforward manner and those that are used for inter-vehicle communications. Specifically covered here are the Naval Undersea Warfare Center's and University of Massachusetts' Common Control Language (CCL), Woods Hole Oceanographic Institution's Compact Control Language (C2L), JAUS, and the Multilateral Interoperability Programme's JC3IEDM.

B. AUTONOMOUS VEHICLE CONTROL PARADIGMS

1. Scripted Control

Of all the methods commonly utilized for autonomous vehicle control, scripted control is the most straightforward. In this methodology, a vehicle mission is defined as a series of discrete commands that can include open-loop commands that order control settings independent from vehicle response (e.g., rudder deflection or power setting), closed-loop commands (e.g., headings, speeds, or waypoints), or commands that order behaviors not directly related to vehicle control (e.g., load a new mission script). Examples of languages used for scripted control of autonomous vehicles include those described in (Brutzman, 94), (Davis, 96) and (Marco, 01) that have been used by the Naval Postgraduate School (NPS) Phoenix and Acoustic Radio Interactive Exploratory Server (ARIES) UUVs, and (Hydroid, 01) for defining Remote Environmental Measuring UnitS (REMUS) UUV missions. Figure 2.1 shows an example Phoenix or ARIES mission using the language described in (Brutzman, 94) and (Davis, 96). In this tasking language, each scripted command appears on a single line, begins with a reserved keyword, and is followed by zero or more (sometimes optional) parameters.

The most significant advantages of scripted autonomous vehicle control are clarity and simplicity. Since the mission consists of a sequentially executed series of steps, it can be intuitively defined in a fairly straightforward manner. Additionally, vehicles executing scripted missions behave in a predictable manner—they execute each command in their script in order until the mission is concluded or a command cannot be completed successfully (in such cases, the most common response is to execute a predefined mission abort script). The most obvious disadvantage to autonomous vehicle control using fixed scripts is the lack of flexibility. The environments in which autonomous vehicles operate are inherently dynamic, making the ability to adapt to changing conditions highly desirable. Additionally, it is highly unlikely that an ideal mission can be defined for anything more than the most mundane tasking using a fixed-sequence script. There exists strong motivation to provide a mechanism by which the vehicle can adjust its mission as more information about the environment is obtained.

While this lack of flexibility limits the capabilities of pure scripting, this control paradigm remains an integral part of many more advanced control architectures that will

be discussed in subsequent sections. One example is provided in the (Nicholson, 04) implementation of the RBM, a three-layer control architecture modeled after the command hierarchy of U.S. Navy submarines (Byrnes, 93)(Healey, et al., 96). At the top (Strategic) RBM layer, an abstract mission definition is utilized to plan a series of script commands that are issued to the middle (Tactical) level for execution. The Tactical level conducts any required numerical processing and provides individual control orders to the lowest (Execution) level which is in turn responsible for actual vehicle hardware interface. In the (Nicholson 04) implementation, the RBM Strategic level generates scripts which are issued to the Tactical level. As the mission progresses, new scripts are generated by the Strategic level and issued to the Tactical level replacing any previously issued script. Viewed by itself then, this RBM Tactical level implementation relies on scripted control. The addition of higher level control structure, therefore, extends the capability of scripted control without eliminating it and provides an example of the natural evolution of robust multi-layer control paradigms from simple scripted control.

```
POSITION 0 0 0
RPM 500
WAYPOINT 100 10 10
HOVER 100 50 10
GPSFIX
RPM 700
WAYPOINT 0 50 5
WAYPOINT 0 100 5
WAYPOINT 100 100 5
HOVER 100 150 10
GPSFIX
WAYPOINT 0 150 10
WAYPOINT 0 200 10
WAYPOINT 100 200 10
WAYPOINT 100 250 10
HOVER 0 250 10
GPSFIX
HOVER 0 0 10 360
DEPTH 0
WAIT 25
QUIT
```

Figure 2.1. A Scripted Phoenix Unmanned Underwater Vehicle (UUV) Mission in the Behavior Scripting Language described in (Brutzman, 94) and (Davis, 96)

Seen against this frequently utilized backdrop, a common autonomous vehicle data model can be applied to the domain of script-based control in a number of ways. The most obvious is the mapping and automatic conversion between task-level behaviors and vehicle-specific script commands. Additionally, a common data model supporting both high-level declarative and task-level mission definition and mechanisms for automatically translating missions of one form to the other can be used to intuitively develop mission scripts that meet declarative mission requirements. More subtle, but possibly more interesting, is the use of a data-model-based planner in place of (rather than as a data source for) upper level of a multi-layer control architecture like the RBM Strategic level. By extension, therefore, this approach provides a pattern for the use of a generic data-model-based planner as a plug-in higher-level controller for arbitrary vehicles that normally run only predefined scripts.

2. Hierarchical Control

A second autonomous vehicle control methodology that is similar in many respects to scripted control is hierarchical control. As with scripted control, a mission consists of a series of steps or tasks. Rather than defining the entire mission in terms of the most atomic steps, a layered approach is utilized where higher layers contain complex tasks and lower layers represent the complex tasks as a series or simpler sub-tasks as depicted in Figure 2.2. Layers at increasing depths of the hierarchy divide complex tasks into increasingly specific sub-tasks. Early hierarchical control exemplars are provided by the Task Decomposition architecture described in (Albus, 93) and the Activity-Based Mission Planning and Plan Management system described in (Hall and Farrell, 94). Both of these systems possess all of the functionality of more modern hierarchical control architectures. As a more specific example of hierarchical planning, a UAV complex task might be "search area X." This task may be divided into subtasks "transit to area X," "anchor in area X," "scan with radar," and "return to base." The "transit to area X" task might be subdivided yet again into executable subtasks "take off," "climb to transit altitude X," and "follow route to area X entry point."

A hierarchical-control mission is specified as a series of complex tasks or prioritized goals. Planning algorithms are applied at each level to generate subtasks for the next lower level. Planning is typically only required for the next task to be executed

at each level. High-level plans, therefore, are more general in nature and cover longer periods of time while low-level plans are increasingly more detailed but are expected to complete in the relatively near future. As a result, effort is not expended in the generation of detailed long-term plans that have a high probability of becoming obsolete before they are fully executed (Stentz, 04).

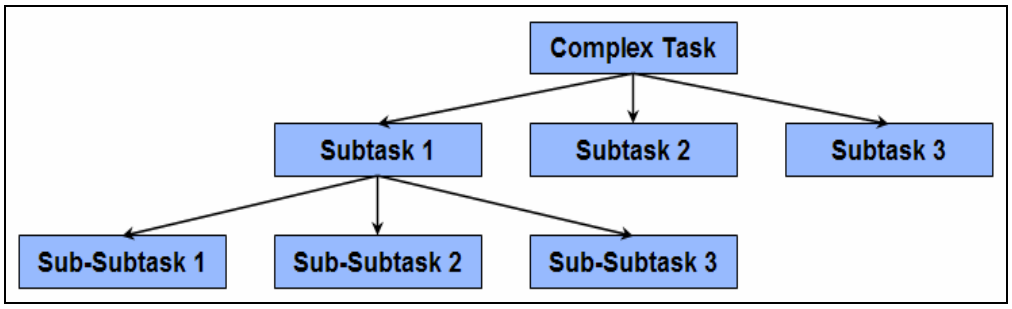


Figure 2.2. A Typical Hierarchical Architecture for Autonomous Vehicle Control

Among the most capable multi-layer control architectures currently available is the Draper Laboratory's All-Domain Execution and Planning Technology (ADEPT) (Ricard and Kolitz, 02). The result of ten years of evolution of autonomy projects such as the Mission Planning and Plan Management system of (Hall and Farrell, 94), ADEPT uses increasingly detailed activities that are drawn from an activity library appropriate to the hierarchical level. Each activity consists of an activity model (i.e., a description of the ultimate effect of the activity) and an activity planner that describes how the activity is to be decomposed into activities appropriate for the next lower level (Hall and Farrell, 94). Path planning algorithms include A*, Focused Dynamic A* (D*), and Time-Bounded A* (T*) search implementations while other activities utilize algorithms more suited to their specific requirements (Ricard and Kolitz, 02). In all cases, monitoring and diagnosis modules at each layer direct planning and execution based on mission requirements and evolving situational awareness.

Hierarchical control has the advantage of being well suited for complex tasks—the types of tasks that can be decomposed into subtasks in a fairly straightforward manner. Numerous planning algorithms have been specifically developed to deal with this sort of problem and can be applied arbitrarily in generating subtasks at various levels. Additionally, recent experimentation with hierarchical control and planning for multi-

vehicle systems provides an indication that the paradigm is well-suited not only for single-vehicle control, but is potentially applicable in the domain of multi-vehicle control as well (Yang, et al, 05). The most significant disadvantage is potentially slow, intermittent, or inappropriate response to a dynamic or uncertain environments due to planning and replanning requirements. Also a factor is the dependence of successful plan execution on the accuracy of the vehicle's world model at the time the plan was developed. (Stentz, 04)(Russell and Norvig, 03)

On a superficial level, the application of a common data model to the domain of hierarchical control is relatively simple. It consists of mappings between the common data model and potential tasks and subtasks coupled with automated translations between the vehicle-specific hierarchical commands and the common data model. A more interesting application of the ontology to hierarchical control arises from its potential to represent both declarative and task-level missions along with the ability to automatically convert between the two. As previously noted, the conversion from data-model-compliant declarative goals to task-level missions closely mirrors a hierarchical control architecture's generation of detailed plans from high-level tasks. The obvious conclusion is that a single data model or ontology can be used as the basis for implementation of a hierarchical control architecture. As alluded to previously, this ontology-based control architecture can provide a simple means of extending the capabilities of simple scripted control architectures by adding more robust high-level control.

3. Behavioral Control

A third method of autonomous vehicle control is behavioral control. Vehicles using this form of control activate predefined behaviors as required to achieve the goals of the mission. Available behaviors for a vehicle might include "maintain heading," "avoid obstacle," or "track target." In most cases, multiple behaviors can be active simultaneously. For instance an UUV might have one behavior controlling heading, another behavior controlling depth, and a third behavior controlling sensors. Many behaviors, on the other hand, are mutually exclusive and cannot be active at the same time—"maintain heading" and "avoid obstacle," for example, are in all likelihood incompatible. It is the responsibility of the top level of the vehicle control architecture to ensure that mutually exclusive behaviors are not activated simultaneously. (Stentz, 04)

In its simplest form, behavioral control mirrors the control paradigm of a purely reactive agent. Consider, for example, a homing torpedo—an autonomous underwater vehicle (AUV) with a simple unchanging goal: impact a target. Immediately following launch, active behaviors might consist of "dive," "power up computer and sensors," and "steer to search bearing." Upon reaching search depth and heading with computer and sensors powered up, "maintain depth," "steer through search pattern," and "search for target" behaviors might become active. Finally, upon target detection, final-phase behaviors such as "attain target depth," "steer towards impact point," and "track target with sensor" can be activated and will remain active until impact.

In more complex systems, behaviors are activated and deactivated based on arbitrary and possibly dynamic run-time conditions rather than a predetermined static mission script. In these systems, a more robust top-level controller is required to activate and deactivate the appropriate behaviors to successfully complete a complex mission in a dynamic environment. One such system is the Distributed Architecture for Mobile Navigation (DAMN) (Stentz, 04). The top-level controller of this system utilizes an arbiter to determine vehicle controller commands based on the currently perceived situation, the requirements of the mission, and the potential control requirements of the various competing behaviors. As the example heading-behavior arbiter depicted in Figure 2.3 indicates, the DAMN arbiter examines the control command called for by all behaviors potentially controlling a specific parameter and then determines the most advantageous behavior to activate for the current situation. That action may be the one that progresses most directly towards the highest priority goal, minimizes the near-term risk to the vehicle, or some combination of these or other desirable outcomes.

Another example of a robust behavior control-based system is The Pennsylvania State University Applied Research Laboratory's Intelligent Control architecture. This system uses a perception module that fuses and interprets sensor data in order to maintain situational awareness and build a comprehensive world view that includes all perceived objects and their classifications. Behaviors are contained within a response module that assumes mission management and planning responsibilities. A mission manager in the response module determines the appropriate behaviors to activate and replans as required when the world model changes or received communications modify the mission

requirements. Active behaviors control actuators, sensors and outbound communications. A graphical depiction of the relationship between the perception and response modules is provided in Figure 2.4. (Lewis and Weiss, 04)

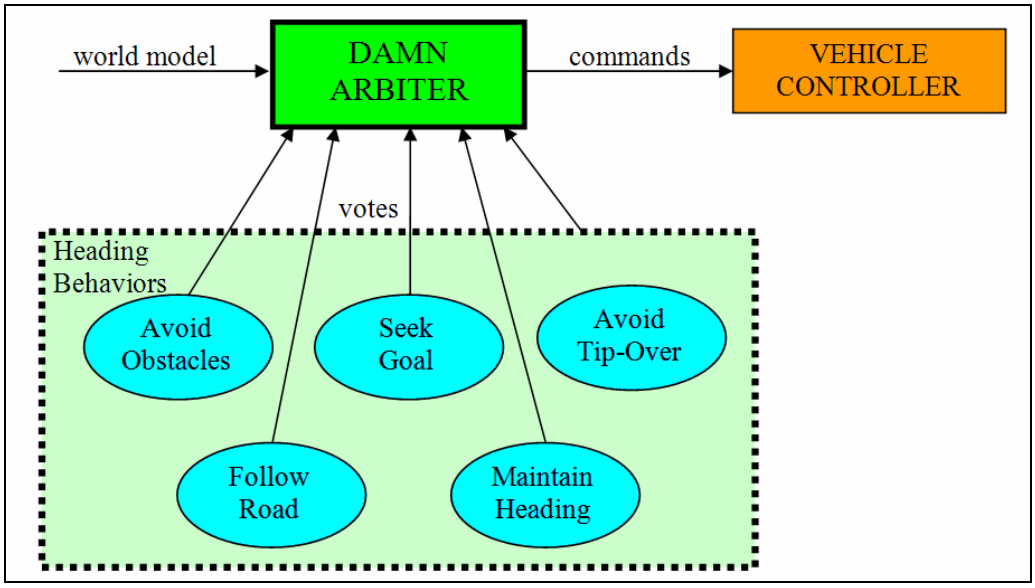


Figure 2.3. A Distributed Architecture for Mobile Navigation (DAMN) Arbiter for Autonomous Vehicle Heading Control Behaviors (After: Rosenblatt, 97)

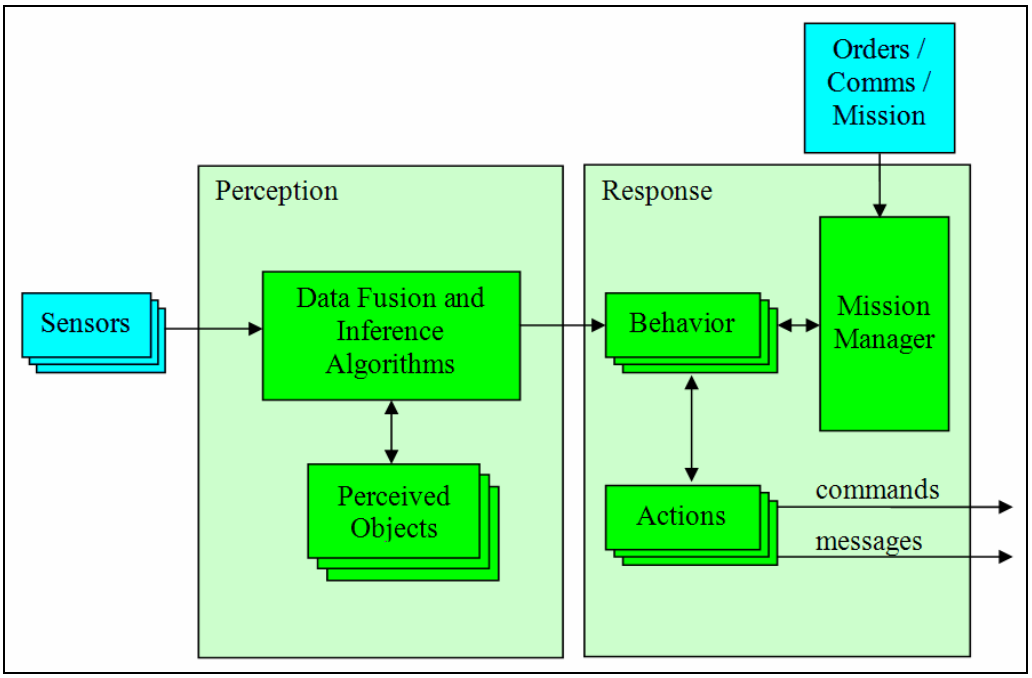


Figure 2.4. Behavioral Autonomous Vehicle Control as Implemented in the Pennsylvania State University Applied Research Laboratory Intelligent Control Architecture (After: Lewis and Weiss, 04)

The most significant advantage of behavioral control is execution speed. Since the system relies only on the current vehicle state and operates without regard to past occurrences or future predictions, planning requirements are minimal. Whereas planning algorithm computations generally run in polynomial time, world-state evaluation and behavior selection can be executed in near-constant time. In addition, behavioral control does not rely on any *a priori* world knowledge—another byproduct of the historical independence. On the down side, behavioral control is not nearly as conducive to complex tasks as hierarchical control. While the specific intent of a planning algorithm is to decompose a complex task into a series of relatively simple steps, it is far more difficult to design behaviors, or more specifically behavior activation criteria, to support tasks of this sort (Stentz, 04). Nevertheless, the ability of vehicles utilizing behavioral control to operate as part of a multi-vehicle system to accomplish complex tasking has been demonstrated and behavior-based control remains an area of interest for autonomous vehicle researchers (Lewis and Weiss, 04).

The observation that behavior initiation for vehicles utilizing behavioral control is not governed by a script might make it appear that the proposed common data model is of limited applicability to these vehicles. However, while the use of task-level behavior scripts is not, on its face, relevant to behaviorally controlled vehicles, a common thread among the more robust behavioral control vehicles is the requirement to specify what a mission is intended to accomplish. It is primarily in this area that the proposed common data model is potentially useful—by providing a tasking specification means for vehicles utilizing any control paradigm, the model serves as a bridge between vehicles regardless of the control paradigm utilized during both planning and execution phases of the mission.

4. Hybrid Control

A final autonomous vehicle control methodology that attempts to capture the advantages of the previously discussed paradigms while mitigating their disadvantages is hybrid control. Whereas hierarchical control is inherently deliberative in nature and behavioral control is inherently reactive, the hybrid control paradigm attempts to combine the best of both by implementing hierarchical control at higher levels and behavioral control at the lowest levels (Russell and Norvig, 03)(Stentz, 04). Higher levels, therefore,

contain global plans consisting of complex tasks and subtasks, while lower levels utilize behaviors that are activated as required for the ordered completion of the ordered subtasks.

Among hybrid architectures, implementations along the lines of the three-layer architecture depicted in Figure 2.5 are the most common (Russell and Norvig, 03). In the most general version of this model, the highest, or deliberative, layer transforms a mission comprised of complex tasks into subtasks (e.g., a series of waypoints) that are sent to an intermediate layer (referred to in the literature as the executive layer) for sequencing and execution. In some cases, the deliberative layer may have multiple sublayers that divide the complex tasks into increasingly simplified subtasks until a plan of sufficient detail for executive layer processing is obtained. The executive layer is responsible for activating and deactivating behaviors at the reactive layer in order to execute deliberative layer directives. Additionally, the executive layer is responsible for interpreting sensor data to maintain a world model that is utilized by the deliberative layer during planning. The reactive layer interfaces with the vehicle's control hardware and implements behaviors that react to the local environment.

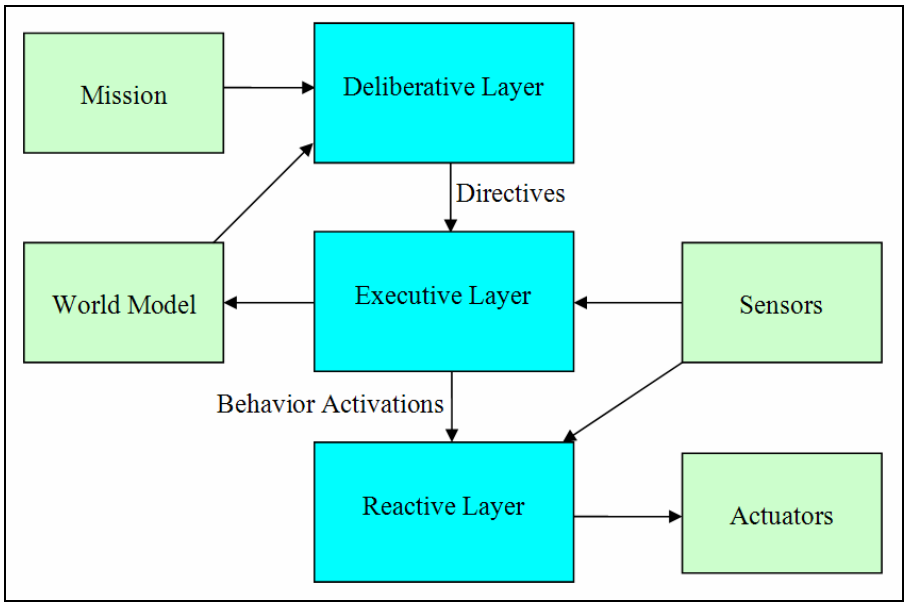


Figure 2.5. A Three-Level Hybrid Architecture for Autonomous Vehicle Control

The most significant advantage of a hybrid architecture is the ability to plan for and execute complex tasks in a dynamic and uncertain environment without sacrificing

the low-level efficiency—deliberative and executive layer computations can proceed at an appropriately slow pace while reactive layer behaviors provide rapid, real-time response to the external environment. In this respect, hybrid control achieves a dual goal of capturing the advantages and eliminating the disadvantages of both hierarchical and behavioral control. This is a significant enough accomplishment that many control architectures initially implemented in a purely hierarchical manner have evolved into hybrid architectures (Albus, 96)(Albus, 98). A hybrid control architecture does, however, retain hierarchical control's dependence on the accuracy of the world model upon which deliberative planning relies. (Stentz, 04)(Russell and Norvig, 03)

A common autonomous vehicle data model can be applied to the domain of hybrid control in essentially the same way that is applied to hierarchical control. At its most basic level, this consists of mappings and conversions between the ontology-compliant data format and the hybrid control complex tasks and subtasks. This similarity stands to reason since hybrid control does not differ substantially from hierarchical control above the executive layer. Also applicable is the use of the data model as an integral part of a multi-layer hybrid control architecture. In fact, the exemplar explored in this research utilizes the exemplar data model in support of a hybrid control architecture—specifically as the interface mechanism between the levels of the RBM.

5. The Rational Behavior Model (RBM)

As stated previously, the RBM is a three-layer hybrid architecture. First proposed in (Kwak, et al., 92) and formalized in (Byrnes, 93), the RBM structure is designed to roughly model the command hierarchy of a naval vessel as depicted in Figure 2.6. The top RBM layer, referred to as the Strategic level, correlates to the vessel's commanding officer and is responsible for high-level decision making and mission flow. The middle layer, the Tactical level, correlates to officer watchstanders (e.g., officer of the deck, navigator, etc.) and is responsible for executing Strategic level directives and monitoring vehicle systems. Finally, the lowest layer, the Execution level, has little decision-making responsibility but provides the interface with the vehicle's control and sensor systems. This level of the architecture correlates to the junior members of the naval vessel watch team.

At the Strategic level, vehicle tasking is expressed as a set of inference rules that describe the goals of the mission and define a search space that breaks the goals into subgoals that directly equate to vehicle behaviors that are issued as orders to the Tacitcal level. The RBM Strategic level, therefore, can be viewed as a theorem prover that executes the specified mission through side effects of the inference process (Byrnes, 93).

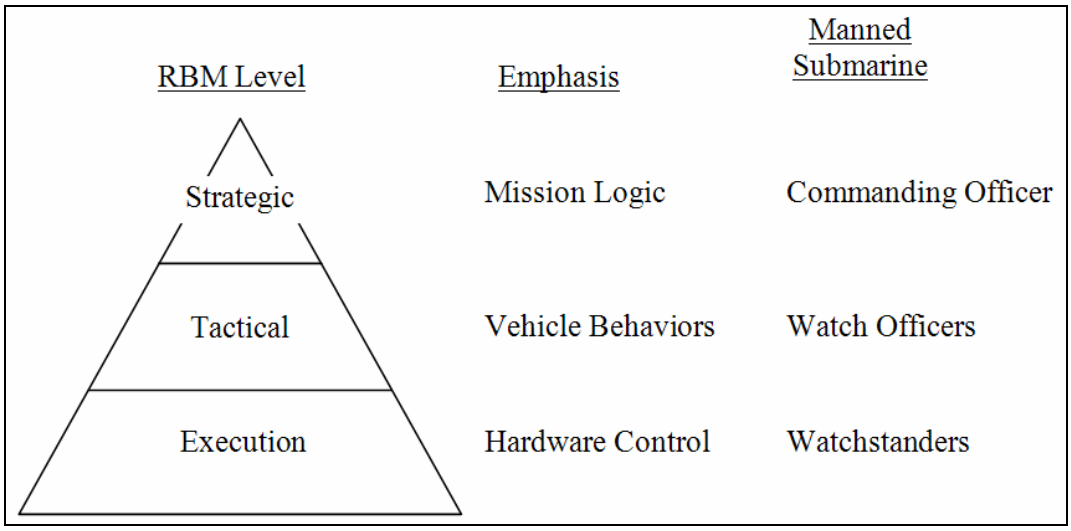


Figure 2.6. The Rational Behavior Model (RBM) Architecture that uses the Control Paradigm of Naval Vessels as its Basis (After: Byrnes, 93)

Alternatively, the Strategic level can be viewed as a finite state machine where individual goals correspond to states and transitions are executed upon success or failure of the corresponding goal. This second view closely matches the declarative mission specification of the exemplar data model developed in the conduct of this research. Similarly, behavior orders issued to the Tactical level equate to data model task-level behaviors. These similarities are exploited in the extension and implementation of the RBM described in Chapter VII.

C. SYSTEM AND PLATFORM-INDEPENDENT LANGUAGES

1. Common Control Language (CCL)

CCL is a research project by the Naval Undersea Warfare Center, University of Massachusetts, and the Autonomous Undersea Systems Institute that is similar to the common data model proposed by this research in scope and intent but differs significantly in implementation. As with the common autonomous vehicle data model developed here, the intent of CCL is to provide a command language suitable for the

control of arbitrary vehicles or systems of heterogeneous vehicles (although CCL is intended only for UUVs). Also like the common data model, CCL constructs are available both for the development of vehicle tasking prior to launch as well as for in-mission inter-vehicle communications.

Building upon previous efforts to develop a common UUV command and control language (Blidberg, 94)(Turner and Chappell, 95)(Buzzell, 04)(Komerska, et al., 99-1), CCL is based on predefined behaviors, each of which fall into one of nine classes (Figure 2.7). A CCL program defines a mission by specifying tasks. Each task is specified with initialization parameters (points, parameters and initial values, authorized behaviors, and available operators), a description of how and when the task is to be updated, a definition of the task cost, and a goal definition for the task. Each task can be defined in terms of subtasks or individual behaviors. A single mission level task statement is used as the top-level mission definition. Since the available behaviors upon which all tasks are ultimately based are vehicle independent, CCL missions can be defined for arbitrary UUVs (Duarte, et al., 05).

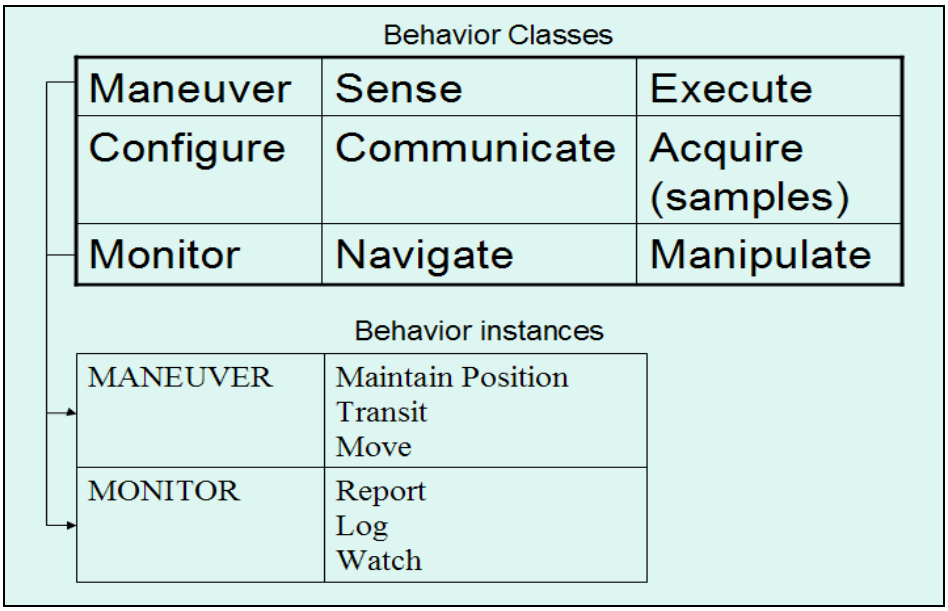


Figure 2.7. Common Control Language (CCL) Behavior Classes and Example Instances (From: Duarte, et al., 04)

Defining the mission, however, is only half the problem—it still has to be able to run on the vehicle for which it is intended. CCL accomplishes this through the on-

vehicle installation of the CCL hybrid controller. This controller consists of a CCL interpreter and an embedded planner and is used to generate vehicle-specific controller commands. The planner, based on a form of process algebra referred to as cost calculus ($-calculus) that is specifically concerned with representing and manipulating concurrent systems in a resource-constrained environment, uses kΩ-optimization to develop appropriate behavior sequences (Eberbach, 01)(Eberbach, 05). A characteristic of kΩ-optimization that makes it attractive for on-vehicle utilization is that it guarantees at least a suboptimal solution regardless of allotted computation time (Duarte, 04).

The CCL controller runs in the Naval Undersea Warfare Center's Distributed Control Environment—a behavior-based software environment based on the University of Southern California's Ayllu system (Werger, 00) for concurrent systems that uses a shared memory structure. The CCL hybrid controller also runs within the Distributed Control Environment and utilizes the behavior specifications and an updated world model (based on sensor data and potential input from other vehicles or human operators) to develop plans that accomplish the tasking (Duarte, et al., 05). The current plan is used to generate native controller commands that maneuver the vehicle and control its sensors as illustrated in Figure 2.8.

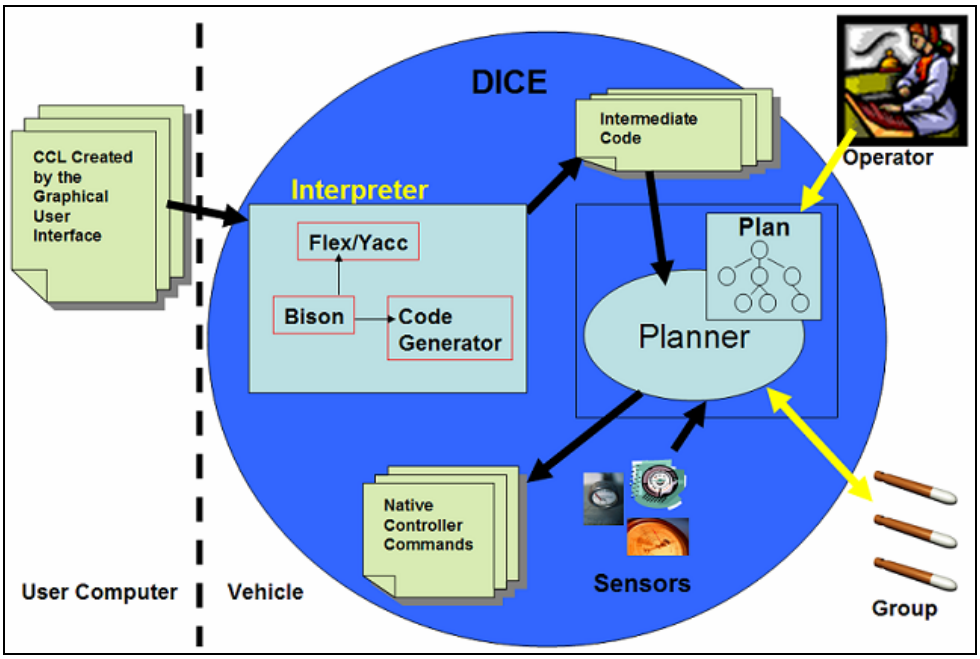


Figure 2.8. The CCL Runtime Environment (After: Duarte, et al., 04)

Communication in CCL builds on (Turner and Chappell, 95) and the Foundation for Physical Agents (FIPA) Communicative Act Library Specification (FIPA, 02). CCL messages fall into one of two categories: request or inform. Request messages are used to issue commands and query other vehicles for information. Inform messages are used to propagate data and knowledge through the system. Messages include header and scheduling information as well as a variable-length body and are designed so that a parser does not need to backtrack. Messages can be used to transmit tasks, behavior definitions or status information from one vehicle to another. (Duarte, et al., 05)

As with programs written in other languages, UUV missions defined with CCL can become quite complex which can make them difficult to read, author and debug. This implies that a graphical user interface for the development and testing of CCL missions might be advantageous and two such graphical user interfaces are currently in development. The first is being developed in conjunction with CCL itself and is designed to generate missions for single vehicles (Duarte, et al., 05). The second, the Autonomous Systems Monitoring and Control system is designed for the development of multi-vehicle missions (Mupparapu, et al., 04). Additionally, the Autonomous Systems Monitoring and Control system provides facilities for communicating with vehicles and monitoring mission progress at run time.

The most significant advantage to CCL is its inherent support for arbitrary UUVs. The predefined behaviors through which arbitrary vehicle support is achieved are conceptually similar to the task-level behaviors of the exemplar data model developed in the course of this research. Additionally, communication and coordination is simplified by the fact that all vehicles utilize the same CCL controller. The main disadvantage is the requirement to install the CCL controller on each vehicle. On-vehicle CCL implementation is simplified through the vehicle-specific implementation of CCL behaviors (i.e., software modules that convert CCL behaviors into vehicle-specific control orders) and a “bridge” behavior that provides an interface between the CCL controller and the vehicle’s existing control software (Duarte, et al., 05).

2. Compact Control Language (C2L)

C2L is a project of the Woods Hole Oceanographic Institution Oceanographic Systems Laboratory that, like CCL and AVCL, is intended to facilitate dissimilar vehicle

interoperability. Its focus, however, differs in that it is concerned primarily with communications—its purpose is to serve as a language between UUVs whose communications paths are generally limited to low-bandwidth acoustic modems. C2L is currently the basis for acoustic communication of the Hydroid REMUS UUVs, but is intended to be generic enough to apply to arbitrary UUVs. C2L is presently mandated as the acoustic communications protocol to be used among UUVs within the Office of Naval Research Very Shallow Water / Surf Zone Mine Countermeasure program (Duarte, et al., 05), and has been implemented on a number of swimming and crawling UUVs in addition to REMUS (Stokey, et al., 05).

At the time of this writing, the current C2L specification consisted of 21 message types (Stokey, 05). The message structure is designed around the capabilities of the Woods Hole Oceanographic Institution acoustic modem set (variants of which are commonly installed on many UUVs), so C2L message size is restricted to the 32-byte packets of the modems. Since C2L is specifically intended for bandwidth-limited communications paths, an effort was made to capture a significant amount of information in the smallest message possible. The 32 bytes of a C2L message packet can contain multiple submessages as shown graphically in Figure 2.9. This is typically accomplished by using simple compression algorithms to on individual data elements. Latitudes and longitudes, for instance, are represented with bytes to within several meters resolution (Stokey, et al., 05). Packet size notwithstanding, C2L message length is not specifically limited to 32 bytes, but while the potential exists for multiple-packet messages, at present none have been implemented (Stokey, 04).

Not surprisingly, C2L is capable of transmitting vehicle status information along the lines of position and heading, environmental sensor information such as bathymetry and salinity, as well as short text or error messages. Additionally, various command and control communications are available to allow run-time mission modification or the issuance of specific control commands (start, abort, etc.). Finally, since substantial Woods Hole Oceanographic Institution UUV research has focused on the utilization of UUVs in mine countermeasure operations, C2L has predefined messages for the transmission of mine countermeasure data and reports. Specifically, C2L defines messages that encapsulate Computer-Aided Detection / Computer-Aided Classification

data. The Computer-Aided Detection / Computer-Aided Classification system is used for automatic detection and classification of mine-like objects based on raw sonar data. It can be installed on a UUV to provide for in-mission classification and decision making or utilized for post-processing of UUV data files (Dobeck, et al., 04).

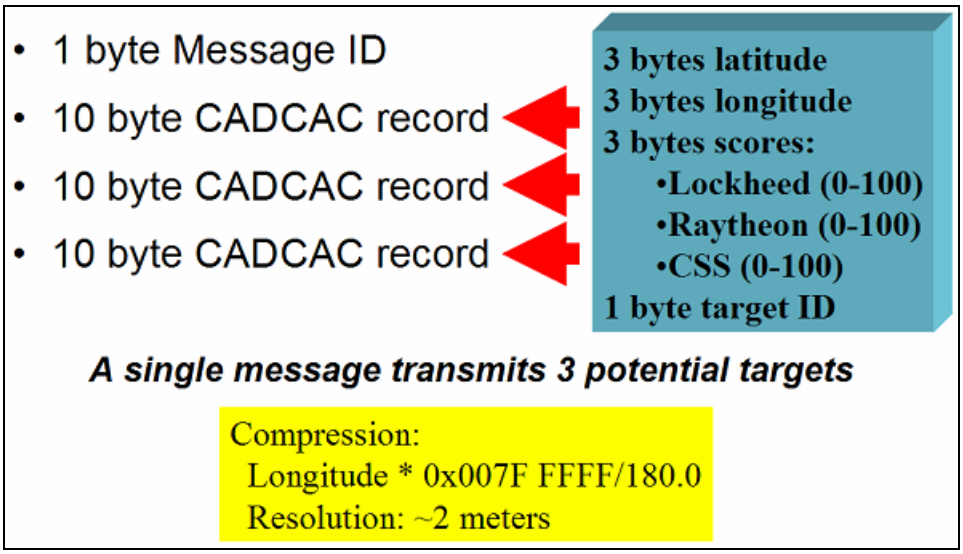


Figure 2.9. A Compact Control Language (C2L) Message Containing Computer Aided Detection / Computer Aided Classification Mine Countermeasures Data (After: Stokey, 04)

A potential disadvantage of C2L is that vehicles must be programmed to utilize it. This inhibits the ability of C2L-capable vehicles to interoperate with vehicles that are not C2L-capable. Additionally, a number of messages in the C2L vocabulary are tailored around the capabilities of specific vehicles and the data that these vehicles collect. The Redirect message, for instance, closely matches the REMUS command directing a lawn-mower-like coverage pattern over a specified survey area. Developers interested in implementing C2L for specific vehicles are required to coordinate with the C2L developers to extend the message set (changes to existing messages are not allowed) to support their specific vehicle requirements (Stokey 05). In order to facilitate the operations of heterogeneous vehicle systems, the C2L specification does not allow "partial compliance"— vehicles that can interpret the entire C2L message set are compliant and vehicles that cannot are noncompliant. It is also assumed that if any participants in a particular operation are using extensions to the existing message set, all participating vehicles will be able to interpret these messages as well (Stokey, 05).

While theoretically suitable for arbitrary autonomous vehicles, these constraints mean that C2L is not inherently applicable outside the set of vehicles that have implemented it. Nevertheless, as with vehicle-specific data formats, the C2L semantics can be captured by a common data model and previously discussed methods of automated translation can be utilized to extend the compatibility of C2L beyond the set of implementing vehicles.

3. Joint Architecture for Unmanned Systems (JAUS)

a. JAUS Overview

Although not a platform-independent language along the lines of CCL and C2L, JAUS provides another example of relevant current research. Acknowledging the advantages of a standard open architecture in the cost-effective development and procurement of unmanned vehicles for military applications, the Joint Robotics Program has listed the definition and evolution of such a system as a priority in its Unmanned Vehicle Master Plan and also endorsed JAUS toward this end (JRP, 04). JAUS provides a framework for the logical organization of vehicle modules and also for how they interact. A JAUS system includes both the system's hardware and software.

Among the stated JAUS objectives are support for all classes of unmanned vehicles and interoperable unmanned systems—two design objectives which this research attempts to address (JAUS, 04-1). Also similar to those of the common autonomous vehicle data model are the philosophical underpinnings of JAUS: platform, mission, computer resource, technology, and operator-use independence (JAUS, 04-1).

b. JAUS System Topology

A JAUS topology is defined in terms of a system, subsystems, nodes, components, and component instances hierarchically arranged as depicted in Figure 2.10. A system is a logical grouping of one or more subsystems that gain a cooperative advantage by being grouped together (JAUS, 04-2). A JAUS system, for example, might consist of an operator control unit for an unmanned system or a mission planning application for an autonomous system, one or more vehicle subsystems, and possibly support subsystems (e.g., signal repeaters).

Each JAUS subsystem is a distinct unit that operates as an independent entity within the framework of the system. An individual vehicle is considered a

subsystem within a JAUS system so it is this level of the topology with which this work is primarily concerned. JAUS messages are the sole means of communication between subsystems.

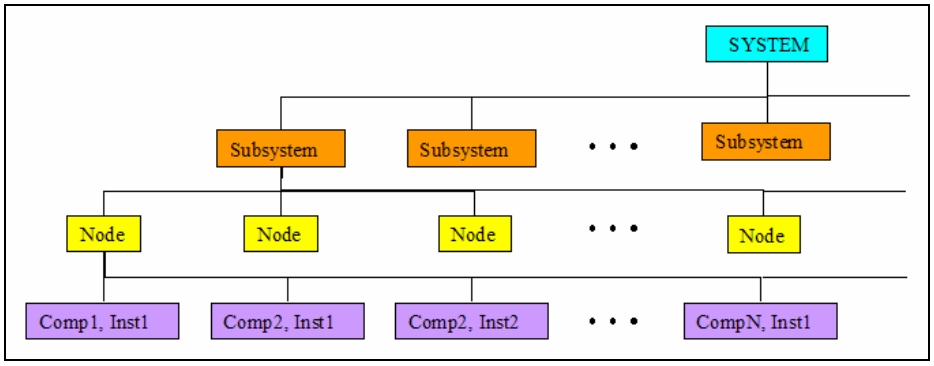


Figure 2.10. Joint Architecture for Unmanned Systems (JAUS) Topology for Unmanned System Design and Implementation (From: JAUS, 04-2)

A JAUS node contains hardware and software necessary to support a single well-defined capability. In general, the division of a subsystem into nodes is up to the designer, but an autonomous vehicle subsystem divides logically into nodes such as master controller, navigator, and vision processor. Each node is logically self contained and includes both the hardware and software required to implement the intended capability. As with subsystems, nodes communicate exclusively via JAUS messages.

Components are the lowest level of the JAUS topology and comprise the software building blocks of nodes, subsystems and systems. Typically implemented as an executable task or process, a component is a software unit that provides one or more services. Unlike the functionality and organization of subsystems and nodes, which are determined by the system designer, the JAUS Reference Architecture defines a fixed set of components that can be used arbitrarily within nodes (including multiple instances of a component) to achieve the desired functionality (JAUS, 04-2).

c. JAUS Components

While the component-level architecture of vehicle systems is somewhat outside the scope of this research, JAUS components and JAUS communications are closely enough related to make a brief discussion of components worthwhile. As stated previously, a component is intended to provide a single cohesive function. Additionally,

components are intended to be self contained in order to minimize communications bandwidth requirements (JAUS, 04-2). Components are divided into five functional groupings: command and control, communications, platform, manipulator, and environmental sensor. A JAUS component specification includes a unique component identification and a functional description (Table 2.1). The component's grouping determines which JAUS messages a component must respond to and in what manner.

Component	ID	Group	Function
System Commander	40	Command and Control	Responsible for overall system control
Subsystem Commander	32	Command and Control	Responsible for overall subsystem control
Communicator	35	Communication	Responsible for all communications into or out of a subsystem
Global Pose Sensor	38	Platform	Maintains the global six degree of freedom posture of the vehicle
Local Pose Sensor	41	Platform	Maintains a local coordinate system posture of the vehicle
Velocity State Sensor	42	Platform	Maintains vehicle linear and angular velocity
Primitive Driver	33	Platform	Controls vehicle motion without external reference
Reflexive Driver	43	Platform	Adds external reference (safety, obstacle avoidance, etc) to primitive driver control
Global Vector Driver	34	Platform	Controls linear and angular velocity relative to a world reference frame
Local Vector Driver	44	Platform	Controls linear and angular velocity relative to a fixed body reference frame
Global Waypoint Driver	45	Platform	Drives vehicle to a global reference frame waypoint
Local Waypoint Driver	46	Platform	Drives vehicle to a waypoint relative to the current vehicle posture
Global Path Segment Driver	47	Platform	Drives the vehicle along a Bezier spline defined in global coordinates
Local Path Segment Driver	48	Platform	Drives the vehicle along a Bezier spline defined in vehicle body coordinates
Primitive Manipulator	49	Manipulator	Controls an articulated manipulator
Manipulator Joint Position Sensor	51	Manipulator	Maintains manipulator rotational and prismatic joint position information
Manipulator Joint Velocity Sensor	52	Manipulator	Maintains manipulator rotational and prismatic joint velocity information
Manipulator Joint Force / Torque Sensor	53	Manipulator	Maintains manipulator rotational joint torque and prismatic joint force information
Manipulator Joint Positions Driver	54	Manipulator	Controls manipulator rotational and prismatic joint positioning
Manipulator End Effector Pose Driver	55	Manipulator	Controls the six degree of freedom posture of a manipulator end effector
Manipulator Joint Velocities Driver	56	Manipulator	Controls manipulator rotational and prismatic joint velocities
Manipulator End-Effector Velocity State Driver	57	Manipulator	Controls the linear and angular velocity of a manipulator end effector
Manipulator Joint Move Driver	58	Manipulator	Controls manipulator joints through a specified path
Manipulator End-Effector Discrete Pose Driver	59	Manipulator	Controls manipulator end effector through a specified path
Visual Sensor	37	Environment Sensor	Controls a vehicle visual sensor (camera, sonar, etc.)
Range Sensor	50	Environment Sensor	Controls a vehicle range sensor (sonar, laser, etc.)

Table 2.1. The Available JAUS Components for use in Implementing Unmanned Vehicle Functionality (After: JAUS, 04-2)

The command and control component group is fairly self explanatory—these components implement the higher-level logic required for mission planning and control of subsystems, nodes or components for which they are responsible. Available command and control components are the System Commander and Subsystem Commander. For obvious reasons, these components are allowed to exchange messages of any type with other components as required. Since it is within these components that higher-level vehicle control is implemented, it is primarily here that the mission-specification aspects of a common data model are most relevant. It is worth noting that JAUS does not specify how a mission is to be represented or what type of control an autonomous vehicle is to utilize. However, it appears that the platform, manipulator and environment sensor components are well-suited to task-level behaviors that have been converted to JAUS messages.

The communications component group consists of a single Communicator component. The role of this component is to provide the single point of communications access to a subsystem. This implies that a JAUS-compliant communications-capable vehicle has a single Communicator component that manages all external data links and communications paths as indicated in Figure 2.11. Subsystem-to-subsystem communication within a JAUS system is unmediated, with each subsystem's Communicator component responsible for processing all received or transmitted messages appropriately. As with command and control components, the Communicator component can exchange any JAUS message with other components as required, but will normally exchange messages only between other components of its own subsystem and Communicator components of other subsystems. Upon message receipt, the communications component of a specific subsystem forwards the information to other components within the subsystem for action as required.

The platform, manipulator and environment sensor components are ultimately responsible for implementing the low-level functionality of any JAUS-compliant vehicle. The available components are listed in Table 2.1 and must be implemented in accordance the JAUS Reference Architecture (JAUS, 04-2). Because of the more limited scope of their functionality, components within these groups have more

limited communications functionality and are required to react in certain ways upon the receipt of certain messages pertaining to their functionality.

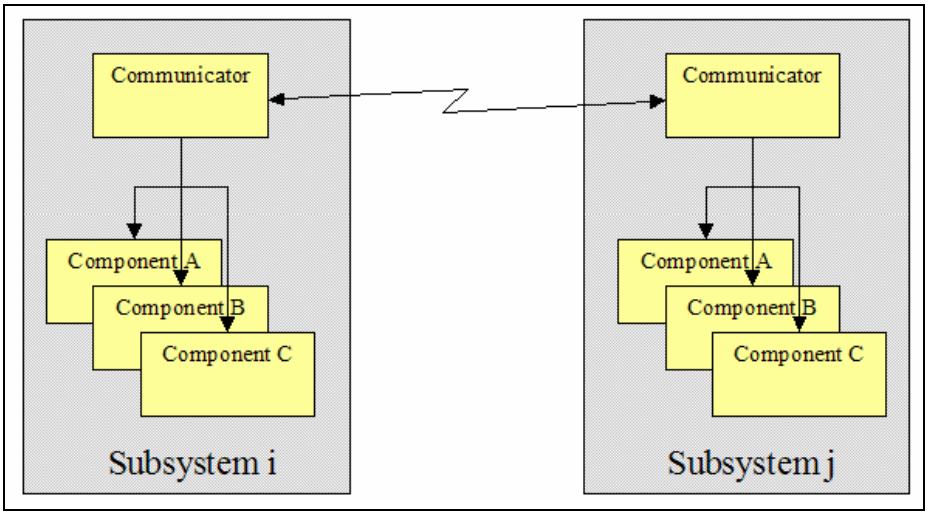


Figure 2.11. The JAUS Communicator Component Functionality at the Architecture's Subsystem Level (From: JAUS, 04-2)

d. JAUS Messaging

When discussing the compatibility of individual vehicles in the context of JAUS, the primary area of concern is messaging—inter-vehicle compatibility from a JAUS standpoint rests on messaging at the subsystem (i.e., vehicle) level and does not rely on JAUS compliance of the internal architectures at the node and component levels. That is, two vehicles are operationally compatible if they can exchange JAUS messages in accordance with the Reference Architecture regardless of whether or not their internal architectures are JAUS compliant. For this reason, the most important aspect of aligning a common autonomous vehicle ontology or data model with JAUS is capture of the relevant messaging semantics. If the data model can be aligned with JAUS, the methods previously discussed can be used to automatically translate between JAUS messages and other message formats to facilitate interoperability of JAUS compliant and non-JAUS compliant vehicles. For this reason, a brief discussion of the format and content of JAUS messages is relevant.

By necessity data formats for use in JAUS messages are explicitly defined (Table 2.2). Additionally, the encoding of many vehicle characteristics is rigidly defined as well, however this does not lead to incompatibility with other systems or with a

common data model utilizing a different encoding. For instance, platform orientation in JAUS is defined by Euler angle rotations about an earth-fixed reference frame (X north, Y east, Z down)—ψ radians about Z, θ radians about Y, and φ radians about X, however this is equivalent to identical rotations performed in reverse order about vehicle body fixed coordinates (X forward, Y right, Z down) and can be equivalently expressed using a quaternion or rotation matrix representations as well (McGhee, et al., 00).

Data Type	Size (in Bytes)	Representation
Byte	1	8 bit unsigned integer
Short Integer	2	16 bit signed integer
Integer	4	32 bit signed integer
Long Integer	8	64 bit signed integer
Unsigned Short Integer	2	16 bit unsigned integer
Unsigned Integer	4	32 bit unsigned integer
Unsigned Long Integer	8	64 bit unsigned integer
Float	4	IEEE 32 bit floating point number
Long Float	8	IEEE 64 bit floating point number

Table 2.2. The JAUS Numerical Data Types (After: JAUS, 04-3)

As with components, JAUS specifies a finite set of available messages and explicitly defines their format, content and meaning. JAUS also provides a means of implementing user-defined messages for application-specific requirements. Available messages have a unique two-byte command code that is included within the message header and fall into one of seven classes (as of version 3.2, JAUS messages are defined for the Command, Query and Inform classes). Each message within a class also has a subgroup designation corresponding to the component group to which the message applies (or core if it applies to all component groups). Messages that are applicable at the JAUS system level are listed in Tables 2.3 and 2.4.

Each JAUS message has a 16-byte header arranged in 12 fields as shown in Figure 2.12. Of these fields the version, service connection flag, experimental flag, data flag, and reserved fields are either fixed or not yet implemented in JAUS. Additionally, since the common data model is concerned with inter-vehicle (subsystem level) as opposed to intra-vehicle (node and component level) communications, the node,

component, and instance portions of the source and destination ID are fixed as well (1, 35 and 1 respectively). Finally, since the maximum length of 4080 for a single JAUS message is sufficient for currently envisioned common data model messages, the sequence number field can also be assumed to be fixed. Of the fields that require consideration during conversion between the common data model and JAUS, the contents of the priority level, acknowledge / no-acknowledge, command code, source ID and destination ID fields are explicitly encapsulated by elements of the common data model and message size can be computed when the JAUS message is constructed. Required and optional data corresponding to each command code is specified in (JAUS, 04-4) and is appended to the end of the JAUS message header.

Message	Code	Subgroup	Description
Set Component Authority	0000h	Core	Sets the component authority relative to others in the system [0..255]
Shutdown	0002h	Core	Shuts down the receiving component
Standby	0003h	Core	Causes the component to suspend operation (if active)
Resume	0004h	Core	Causes the component to resume operation (if suspended)
Reset	0005h	Core	Causes the component to reinitialize
Request Component Control	000Dh	Core	Sender is asserting authority over the receiver
Release Component Control	000Eh	Core	Sender is releasing authority over the receiver
Confirm Component Control	000Fh	Core	Sender accepts or refuses to grant requested control to receiver
Set Time	0011h	Core	Sets the current time and date
Set Data Link Status	0200h	Communications	Enables or disables external data links
Set Wrench Effort	0405h	Platform	Sets vehicle propulsive and / or braking effort for up to six degrees of freedom
Set Global Vector	0407h	Platform	Sets the commanded vehicle speed, altitude, and posture
Set Travel Speed	040Ah	Platform	Sets the commanded vehicle forward speed
Set Global Waypoint	040Ch	Platform	Commands one or more waypoints (latitude, longitude and elevation)
Set Joint Positions	0602h	Manipulator	Commands revolute and prismatic joint settings for a manipulator
Set Joint Velocities	0603h	Manipulator	Commands revolute and prismatic joint velocities for a manipulator
Set End Effector Pose	0605h	Manipulator	Commands a manipulator end effector position and orientation
Set End Effector Velocity State	0606h	Manipulator	Commands a manipulator end effector angular and linear velocity
Set Camera Pose	0801h	Environment	Commands an orientable sensor's posture for up to six degrees of freedom

Table 2.3. The JAUS Command Class Messages for Directing Unmanned Vehicle Actions (After: JAUS, 04-2)

Query and Inform Messages	Code	Subgroup	Description of Data Requested
Query Component Authority Report Component Authority	2001h 4001h	Core	Component's currently assigned authority [0..255]
Query Component Status Report Component Status	2002h 4002h	Core	Vehicle's current operational status
Query Time Report Time	2011h 4011h	Core	Current timestamp
Query Data Link Status Report Data Link Status	2200h 4200h	Communications	Status of external communications links
Query Heartbeat Pulse Report Heartbeat Pulse	2202h 4202h	Communications	External communications check
Query Platform Specifications Report Platform Specifications	2400h 4400h	Platform	Vehicle characteristics (max, min velocity, etc.) breakdown
Query Global Pose Report Global Pose	2402h 4402h	Platform	Vehicle's current latitude, longitude, and altitude
Query Velocity State Report Velocity State	2404h 4404h	Platform	Vehicle's current linear and angular velocity
Query Wrench Effort Report Wrench Effort	2405h 4405h	Platform	Vehicle's current propulsive and braking level of effort in six degrees of freedom
Query Global Vector Report Global Vector	2407h 4407h	Platform	Vehicle's current speed altitude and posture
Query Travel Speed Report Travel Speed	240Ah 440Ah	Platform	Vehicle's current forward speed
Query Global Waypoint Report Global Waypoint	240Ch 440Ch	Platform	Currently commanded waypoint list
Query Manipulator Specifications Report Manipulator Specifications	2600h 4600h	Platform	Number of joints, link lengths, twist angles, offset or joint angles, min and max values for joints of a manipulator
Query Joint Positions Report Joint Positions	2602h 4602h	Manipulator	Current values of the manipulator joints
Query Joint Velocities Report Joint Velocities	2603h 4603h	Manipulator	Current velocities of the manipulator joints
Query Tool Point Report Tool Point	2604h 4604h	Manipulator	Current position and orientation of manipulator end effector
Query Camera Pose Report Camera Pose	2800h 4800h	Environment	Current sensor's posture in up to six degrees of freedom
Query Relative Object Position Report Relative Object Position	2802h 4802h	Environment	Range bearing and elevation (relative to vehicle) of a sensor contact
Query Image Report Image	2807h 4807h	Environment	Raw sensor data

Table 2.4. The JAUS Query and Inform Class Messages for Requesting and Providing Unmanned Vehicle State Information (After: JAUS, 04-2)

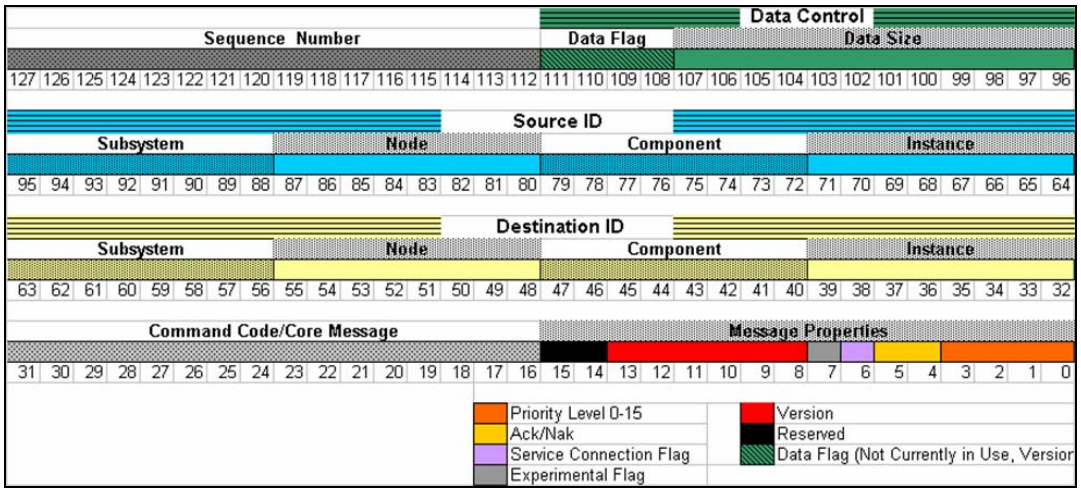


Figure 2.12. JAUS Message Header Layout and Field Descriptions (From: JAUS, 04-3)

e. JAUS Summary

The purpose of JAUS is to provide an open architecture for the efficient design and implementation of unmanned systems. It defines a component-based message-passing architecture that system designers can utilize to build unmanned and autonomous systems with improved interoperability. The portion of the JAUS Reference Architecture most relevant to the development of a common data model is the finite set of rigorously defined messages used to communicate between vehicles.

Although the focus of this section has been on what JAUS provides, also noteworthy is what it does not provide. Specifically, while JAUS specifies the format of individual commands (JAUS messages), it does not specify a format for a complete mission specification. For unmanned systems, which to date make up the majority of fielded and planned JAUS systems, there is no requirement for a mission specification *per se* because the human operators directly control the system in accordance with their understanding of what the mission is supposed to accomplish. In an autonomous vehicle system, on the other hand, the Subsystem Commander component manages mission progress with little or no human intervention. Individual commands, therefore, are likely insufficient unless organized into a complete script. Additionally, since the JAUS message set does not contain messages capable of commanding behaviors any more robust than waypoint transit, JAUS is not inherently suitable for any autonomous vehicle utilizing behavioral, hierarchical or hybrid control.

4. Joint Command Control and Communications Information Exchange Data Model (JC3IEDM)

A conceptually broader common data model is the Multilateral Interoperability Programme's JC3IEDM. The Multilateral Interoperability Programme, currently a voluntary collaboration composed of 26 nations, North Atlantic Treaty Organization Data Administration Group, and Allied Command Transformation, is developing information standards, processes and protocols necessary for international interoperability of command and control information systems (MIP, 03-2). The centerpiece of Multilateral Interoperability Programme efforts to date is the JC3IEDM which defines standard elements of information that are passed between systems. Although the Multilateral Interoperability Programme goals are significantly broader in scope than those of a

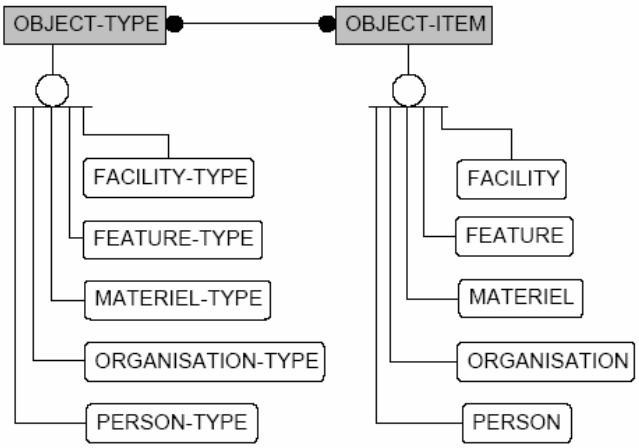
common autonomous vehicle data model, noteworthy similarities exist. Specifically, both attempt to facilitate interoperability of dissimilar systems through data standardization. Further, much information upon which command and control systems rely closely mirrors that utilized by autonomous vehicles in both military and non-military settings. In particular, the purpose of declarative mission definition using the common autonomous vehicle data model is to specify the vehicle that is to be tasked, where it is to operate and what it is expected to do (and not do)—concepts that are rigorously captured by JC3IEDM. Inter-vehicle communications also deals with who, what and where concepts that are central to JC3IEDM, particularly in event and contact reporting. A potential conclusion to be drawn from this data similarity is that JC3IEDM compatibility might be a desirable characteristic in a common autonomous vehicle data model since it leverages the rigor of the existing data model and provides compatibility with current and planned command and control systems.

As stated previously JC3IEDM is specifically designed to encapsulate data that is transferred between command and control systems. Unlike JAUS, which explicitly defines its entire messaging protocol, JC3IEDM is built around the organization of essentially arbitrary data. JC3IEDM defines a relational data model wherein command and control information is expressed in terms of entities and their relationships. The model minimizes the ambiguity inherent in free form text by providing a fixed set of enumerations for many data fields.

JC3IEDM can be described at three levels of abstraction. The conceptual data model (with which this section is concerned) represents generalized concepts such as actions, organizations and locations (MIP, 03-2). Increasing levels of detail are provided by the logical data model which is concerned with entity attributes and the structure of relationships and the physical data model which deals with the specific implementation of compliant systems (MIP, 03-1).

In JC3IEDM entities are characterized as both object-type and object-item. An object-type, as the name implies, is a generalized concept denoting a class of objects—a Predator UAV is an example of an object-type entity. An object-item, on the other hand, is a specific instance of an object-type, Predator UAV bureau number 165275 for

example. Object-types and object-items can be subdivided into the five sub-entities shown in Figure 2.13. Each of these is further divided at least one more time into entities such as aircraft-type (under material-type) or airfield (under facility). Objects can be assigned relationships to other objects along the lines of belongs to, uses or is constrained by via the object-type-establishment and object-item-association entities. Additionally, each object can be assigned capabilities and status as required.



Entity	Entity Definition
FACILITY	An OBJECT-ITEM that is built, installed, or established to serve some particular purpose and is identified by the service it provides rather than by its content.
FACILITY-TYPE	An OBJECT-TYPE that is intended to be built, installed or established to serve some particular purpose and is identified by the service it is intended to provide rather than by its content. Examples include a refuelling point, a field hospital, a command post.
FEATURE	An OBJECT-ITEM that encompasses meteorological, geographic, and control features of military significance.
FEATURE-TYPE	An OBJECT-TYPE that encompasses meteorological, geographic, and control features of military significance. Examples include a forest, an area of rain, a river, an area of responsibility.
MATERIEL	An OBJECT-ITEM that is equipment, apparatus or supplies without distinction as to its application for administrative or combat purposes.
MATERIEL-TYPE	An OBJECT-TYPE that represents equipment, apparatus or supplies of military interest without distinction to its application for administrative or combat purposes. Examples include ships, tanks, self-propelled weapons, aircraft, etc., and related spares, repair parts, and support equipment, but excluding real property, installations, and utilities.
ORGANISATION	An OBJECT-ITEM that is an administrative or functional structure.
ORGANISATION-TYPE	An OBJECT-TYPE that represents administrative or functional structures.
PERSON	An OBJECT-ITEM that is a human being to whom military significance is attached.
PERSON-TYPE	An OBJECT-TYPE that represents human beings about whom information is to be held.

Figure 2.13. A Diagram of the Joint Command Control and Communications Information Exchange Data Model (JC3IEDM) Conceptual Model Object and Object Type (From: MIP, 03-2)

The basic entity for specifying a location in JC3IEDM is a point, which can be specified in either absolute terms (latitude and longitude) or relative to another point. The location portion of the JC3IEDM location entity structure (Figure 2.14) is more or less independent from other portions, the exception being a potential one-to-many relationship with object-item entities. Points can be combined to define line segments, polygons and some surfaces, or utilized to implicitly define other surfaces and most volumes. A point is also the basis for coordinate-system definition—particularly relevant to the development of a common autonomous vehicle data model since it must support vehicles whose position is maintained in relative terms as well as those utilizing absolute positions.

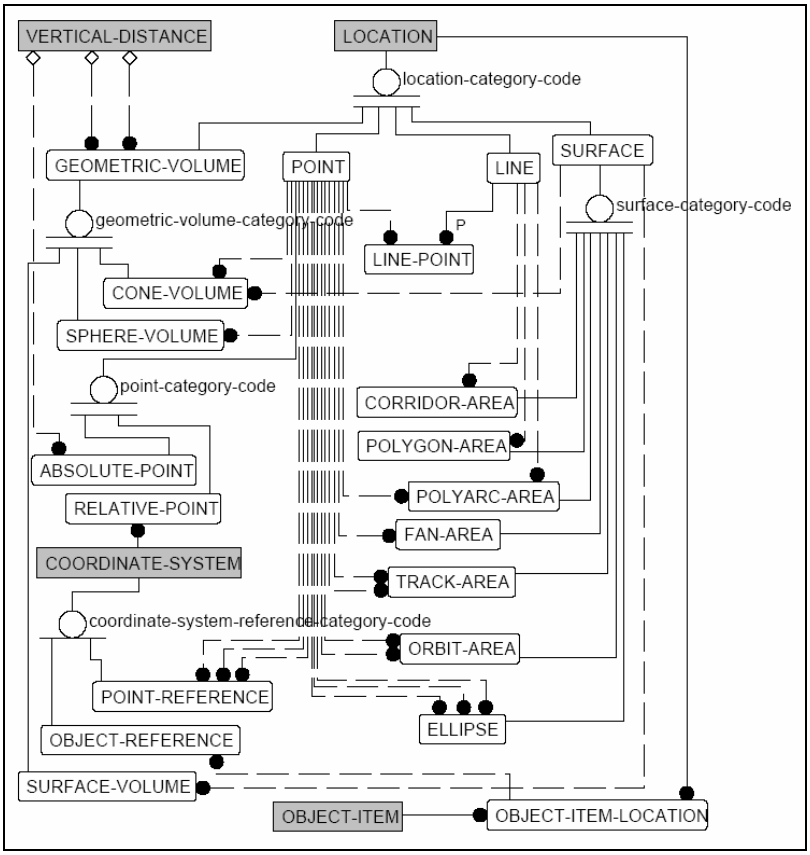


Figure 2.14. A Diagram of the JC3IEDM Location Conceptual Model (From: MIP, 03-2)

The portion of JC3IEDM that is most relevant to the development of a common autonomous vehicle data model is the methodology for specifying actions, since this is central to the specification of mission requirements in a manner appropriate for the

application of planning algorithms. The basic structure of a JC3IEDM action is depicted in Figure 2.15. Action-functional-association entities are utilized to specify sub-actions, define dependencies, specify alternative actions and similar relationships between actions. Action-temporal-association entities specify temporal relationships between actions—when one action can be executed relative to another's execution. Action-objective entities are utilized to specify what objects make up the objectives of the action.

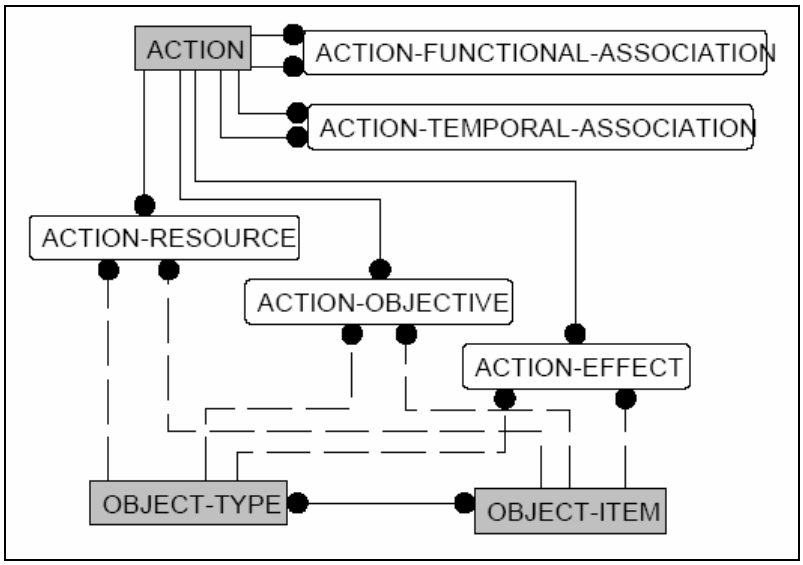


Figure 2.15. A Diagram of the JC3IEDM Action Conceptual Model (From: MIP, 03-2)

This cursory examination of a selected subset of JC3IEDM demonstrates the inherent similarities between many aspects of JC3IEDM and a suitable common autonomous vehicle data model. Subsequent chapters provide a more detailed description of how this similarity can be leveraged to support interoperability between autonomous vehicles and command and control systems.

D. SUMMARY

Various software architectures are currently utilized for autonomous vehicle control. Examples include scripted control, hierarchical control, behavioral control, and hybrid control. This research demonstrates how a common autonomous vehicle data model can be designed in such a way as to be compatible with any of these and can be used to improve the interoperability of vehicles with dissimilar control architectures by enabling mission definition in the same manner regardless of vehicle control architecture.

Interestingly, there have been substantial recent research efforts in various areas with goals similar to those of the common autonomous vehicle data model described here. In particular CCL and C2L are being developed to support interoperability of dissimilar UUVs, JAUS is being proposed as a design and implementation standard for military unmanned vehicles, and JC3IEDM is attempting to standardize data relationships inherent in command and control systems. A common autonomous vehicle data model both complements and is complemented by these evolving technologies. Many characteristics of JAUS and JC3IEDM in particular, are central to the design of the exemplar data model and provide a mechanism through which systems utilizing these methodologies can operate effectively with systems that do not.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXTENSIBLE MARKUP LANGUAGE (XML) AND APPLICABLE XML TECHNOLOGIES

A. INTRODUCTION—WHY XML?

Existing vehicle-specific and general-purpose autonomous vehicle data formats and languages take a variety of forms. Most languages and data formats designed for autonomous vehicle tasking are text based and can range in complexity from fixed-format number sequences where each number's meaning is implied by its location in the sequence (e.g., NPS ARIES UUV waypoint lists) to more robust grammars along the lines of the scripting language of the Hydroid REMUS UUVs. Autonomous vehicle data formats intended for use in communications are more likely to be implemented as bit-mapped binary messages. C2L and JAUS, for instance, are implemented using fixed and variable length binary data packets respectively. Languages such as CCL which can be utilized for both communications and pre-mission tasking can potentially utilize either text or binary formats, however the exemplars discussed here are all text based.

The data model implemented in the course of this research is fairly unique in that it is implemented with XML. (Hawkins and Van Leuvan, 03) makes a strong case for the development and use of XML for autonomous vehicle tasking and interaction. The authors base their argument primarily on XML's platform independence and the ability to translate XML to vehicle-specific formats using XSLT. (Neushul, 03) proposes the use of XML in a broader array of military command and control applications and implements a number of exemplars that rely heavily on XSLT to format data for disparate applications. Included among the exemplars of that research is a proposed XML schema and XSLT-based approach to UAV cooperation.

The proposed use of XML for the autonomous vehicle data model is not surprising given the increasing use of XML in a broad array of applications. In fact, the availability of numerous XML APIs, utilities, and tools has facilitated the development of applications that completely insulate the end user the XML content itself. The end-product data transparency provided by these applications has an important implication for autonomous vehicle systems. Specifically, XML enables the development of vehicle support systems that do not require any level of programming proficiency. Systems

along the lines of the AUVW described in Appendix B, for instance, provide for the development and analysis of vehicle data by non-programmers and autonomous vehicle novices. This capability is an important hurdle in the development of deployable military, commercial and scientific autonomous vehicle systems since it allows vehicle operation by experts in mission requirements and doctrine without the assistance of autonomous vehicle or programming experts.

Ultimately, five specific aspects of XML that are directly or indirectly addressed in these references make a compelling case for its use in the development of AVCL—human and machine readability of instance documents, explicit structure and content governance, document validation, automated conversion between AVCL and other data formats, and the existence of standards and utilities that facilitate the implementation and use of AVCL. The remainder of this chapter discusses each of these in more detail by providing an overview of XML and its use in the development of AVCL.

B. XML OVERVIEW

XML was developed under the auspices of the World Wide Web Consortium (W3C) to foster data exchange over the world wide web (the specific goals of the XML language design team are listed in Table 3.1). A descendant of the Standard Generalized Markup Language (SGML), XML is a metamarkup language (i.e., a markup language without a fixed set of tags) that provides a rigorous means of adding descriptive information (i.e., meta data) to data in order to improve its readability and portability. XML allows application designers to define tags and attributes for the domain of interest that can be used to annotate data in a platform and application-independent manner. This marked up data can then be parsed by standardized XML utilities for use in arbitrary applications. Thus, XML itself is a language for writing other languages, especially data-oriented languages.

The obvious advantages offered by the possibility of platform-independent, self-describing data have led to much hype and speculation concerning the ability of XML to revolutionize computing as we know it. A bit of reality, however, is in order—XML is not a panacea. It is important to recognize not only what XML is, but what it is not. XML, for instance, is not a programming language (although it can be used to implement programming languages). Generally speaking, XML documents do not dictate actions,

they simply provide data to an application in a formal and predictable way. It is up to the application designer to determine how to process the data. XML is also not a network protocol, despite its ubiquitous use for data transfer on the internet. Finally, XML is not a database and it is unlikely that it will replace more traditional database applications. In short XML marks up data to make it more usable, but it does not inherently dictate what to do with it, where it came from, who to send it to, or how to store it. If these limitations are kept in mind, however, XML provides a powerful data management tool that can be applied in a variety of ways. (Harold and Means, 02)

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs that process XML.
5. The number of optional features of XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be humanly legible and reasonably clear.
7. The design of XML shall be formal and concise.
8. XML documents shall be easy to create.
9. Terseness in XML markup is of minimal importance.

Table 3.1. Extensible Markup Language (XML) Design Goals (From: W3C, 04)

Although it was specifically designed to be the language of the web, the applicability of XML extends well beyond the confines of web-based applications. Given the desirability of cross-application data sharing, it is not surprising that XML has become an integral part of applications in domains too numerous to count. In relation to this research, many of the XML design goals of Table 3.1 dovetail nicely with design goals of AVCL. Specifically, the ability to support a variety of applications, ease of processing, human legibility, and ease of document generation are implicit requirements of a common autonomous vehicle data model. Of the remaining design goals, only the explicit lack of any requirement for terseness seems contradictory to the goals of AVCL due to the potential impact of bandwidth limitations on communications between and with autonomous vehicles. Even this pitfall can be dealt with through the use of binary or compressed XML as described later in this chapter.

XML encodes data in the form of a tree. Although the nature of the links between tree nodes is not explicitly mandated, the relationship implied by the structure of an XML document is composition; that is, the children of an XML element comprise all of the

sub-components of that element. The XML fragment of Figure 3.1, for instance, illustrates a potential XML encoding of a UUV waypoint command. The waypoint consists of a two-dimensional Cartesian point (which in turn consists of an X and a Y coordinate), a depth and a speed. In this example the individual data values are expressed as element values of the tree's leaf nodes. Stemming from the concept of using XML to "mark up" the actual data, this pattern is the more traditional and most common method of expressing data in an XML document.

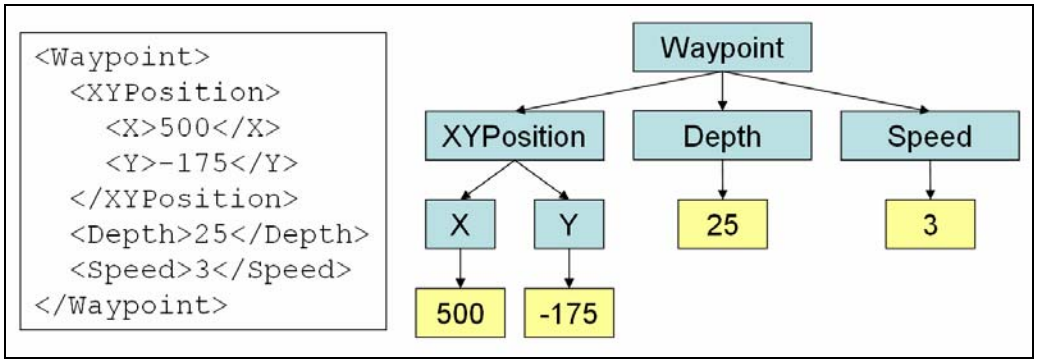


Figure 3.1. An Unmanned Underwater Vehicle (UUV) Waypoint Encoded in XML using Element Values to Capture Data Values

Alternatively, data can be represented in an XML document using element attributes as depicted in Figure 3.2. Traditionally, attributes have been used to provide information about an XML element or its contained value (e.g., in the example of Figure 3.1, attributes might be used to indicate units of measure) but not to express the actual data of interest. In some applications, however, the XML is a part of rather than a description of the data (e.g., the XML fragments of Figures 3.1 and 3.2 are self-contained waypoint commands as opposed to descriptions of the contained numerical data). In cases meeting this criteria, the use of attributes rather than element values to express data is appropriate. XML Schema and the Extensible Three-Dimensional web graphics (X3D) language (ISO and IEC, 04) , for instance, are widely accepted XML languages that utilize this pattern.

Ultimately, the decision of whether to use attributes or element values is somewhat arbitrary. In the case of AVCL, the decision was based on the nature of the encoded data, readability, document size and ease of processing. Additionally, XML data

binding and the heuristics by which data is bound to programming objects also weighed heavily in the decision to use attributes rather than element values.

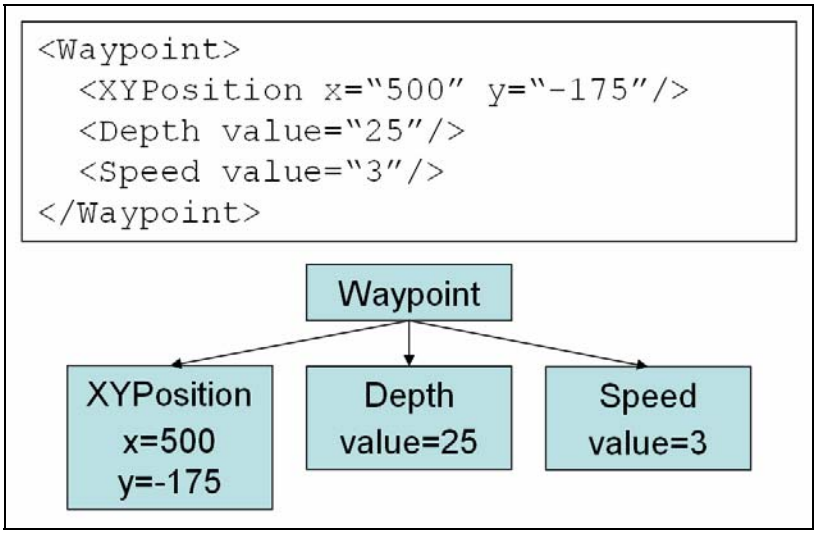


Figure 3.2. An Alternative XML Encoding of a UUV Waypoint with Data Values Expressed using Attributes

C. XML SCHEMA AND DOCUMENT VALIDATION

Beyond requiring a document to be well formed (i.e., all elements require start and end tags and all child elements must start and end inside the parent's start and end tags), the XML specification (W3C, 04-1) places very few constraints on the structure and content of an XML document. Element and attribute names are completely arbitrary (so long as XML naming conventions are observed), the document's content tree can be composed in any manner, elements can have any number of attributes, and elements and attributes can be assigned any numerical or string value. While human operators might reasonably decipher unconstrained XML documents when the element and attribute names are well chosen, computational understanding of unconstrained XML is more difficult. Advances in inference techniques and natural language understanding notwithstanding, the task of programmatic interpretation of XML documents can be greatly simplified if the document structure, content and semantics are known in advance. Specifically, it is desirable to explicitly predefine aspects of the XML documents with which the applications are to work (Duckett, et al., 01) including:

- Elements that can appear in the document.

- Attributes that can appear in the document, to which elements they apply, and whether or not they are required.
- Which elements are child elements and which elements are potential root elements.
- The type, number and order of child elements for each parent element.
- Data types and range restrictions for element and attribute values.
- Default and fixed values for elements and attributes.

In many cases an informal agreement on the XML structure and content is sufficient. This is often true of XML that is intended for a single application or application family when broader use of the XML is not anticipated. On the other hand, when wide use is anticipated, a published standard is often necessary. XML Schema, XSLT, X3D and most ontology description languages fall into this category. However most XML applications, including the data model investigated by this research, fall into a category somewhere between single-application and universal use. AVCL is potentially applicable to a broad enough set of applications that an informal agreement on its content and structure is insufficient, but its scope is narrow enough that a formal specification is unduly cumbersome (it is worth noting that a formal specification is implicitly required for all of the non-XML vehicle-specific languages investigated as part of this research). In short, although a formal standard is not specifically required, the application of a common data model to autonomous vehicle command and control does require a formal mechanism for ensuring data correctness (both syntactic and semantic) and document consistency.

Two mechanisms are available for programmatic constraint of the content and structure of an XML document. XML initially relied on Document Type Definitions (DTD). The main components of a DTD are a set of production rules that define all of the permissible named elements and their content (i.e., child elements, text, etc.) and a set of attribute lists delineating the attributes for each named element. Element production rules are similar in style and functionality to the production rules commonly used to define context-free grammars. Attribute lists, on the other hand, consist of the name of the element to which the list applies and the attribute names, types and optional qualifiers (e.g., to indicate that the attribute is optional or has a fixed or default value). (Hunter, et al., 04)

Despite the fact that DTDs are still in common use, there are a number of shortcomings that make them less than ideal for many XML applications (Duckett, et al., 01). Among the DTD shortcomings are a small set of available data types that do not map to data types of common programming languages and databases (e.g., floating point and integer types), lack of an XML-based syntax, and cumbersome mechanisms for re-use of existing markup constructs. Recognizing these shortcomings, the W3C developed and published a recommended standard for XML Schema—a more robust method of specifying the content and structure requirements of XML documents (W3C, 04-2)(W3C, 04-3). Unlike DTDs, a schema is composed with XML making it compatible with existing XML utilities. Further, XML Schema is significantly more expressive than DTDs, albeit at the cost of increased complexity. The increased power of XML Schema enables designers to utilize more advanced techniques in specifying a content model and also provides more control over document structure. XML Schema directly addresses many of the shortcomings of DTDs and possesses a number of advantages that argue for its use in developing a common autonomous vehicle data model (Duckett, et al., 01):

- Because schemas are written in XML syntax, they can be edited and processed using any tool intended for use with XML documents.
- XML Schema directly supports most primitive data types used in common programming languages and databases.
- XML Schema allows the definition of complex datatypes that extend or constrain existing types.
- XML Schema contains class and type constructs that support re-use, extension, and inheritance of existing markup constructs.
- XML Schema is more expressive than DTDs in constraining mixed content elements (i.e., elements that can potentially contain text).

The advantages of XML Schema over DTDs are substantial enough that U.S. Department of the Navy has recently directed that the use of schemas vice DTDs is required for all forthcoming XML vocabularies and the conversion of existing DTD-constrained vocabularies to XML Schema (DON, 05).

Once authored an XML schema or DTD is normally placed online at a well-known location. Alternatively, a schema or DTD can be maintained locally if the application does not have network access to an online copy as is often the case with autonomous vehicles. Applications utilizing schema or DTD-constrained XML

documents automatically validate instance documents against the schema or DTD when they are loaded. This capability is particularly important for applications that rely heavily on data correctness and for which the consequences of invalid data are potentially severe. autonomous vehicles clearly fall into this category—the time to recognize an invalid document (i.e., an incorrect mission definition or message) is when it is loaded, not after the vehicle has commenced a mission and the invalid data might result in operational errors or vehicle loss.

Most applications and utilities that work with XML documents, including the utilities described in this chapter upon which the proposed common autonomous vehicle data model relies, automatically validate schema-governed documents as they are loaded. The use of XML Schema, therefore, makes documents essentially self validating. Coupled with the self-describing nature of a well-designed XML tag set, automatic validation provides strong incentive for its use as the design mechanism for the common autonomous vehicle data model. AVCL, therefore, has been formally specified using XML Schema. An overview of the data model design is provided in Chapter IV and a complete description of the implementation is provided in Appendix A.

D. XML PARSING

1. Introduction

A number of mechanisms for parsing XML documents are available to XML application programmers, usually at no cost. In general parsers are capable of processing arbitrary well-formed XML documents and do not rely on a schema or DTD. For documents that are governed by a schema or DTD, parsers can validate documents as they are loaded depending on the parser's settings. Further, if an invalid document is encountered, a validating parser is generally capable of identifying the location and nature of invalid content. Parsers do not, however, interpret the document, so it is incumbent upon the application programmer to process the parsed data appropriately.

The most common XML parsers provide API bindings for the Document Object Model (DOM) and the Simple API for XML Parsing (SAX). While neither was utilized extensively in the conduct of this research, they rate a brief description here for two reasons. First, their widespread use and availability in most programming languages make DOM and SAX potential candidates for XML parsing in applications that use the

common autonomous vehicle data model, including vehicles themselves. Second, DOM and SAX parsers form the underpinnings of many of the more advanced utilities upon which this research relies. The mechanics of DOM and SAX, therefore, influence what these utilities can do and how they do it. At the very least they provide straightforward examples of the two prevailing XML processing paradigms: use of the parser to develop a parse tree that is retained in memory indefinitely, and use of the parser to trigger events as it traverses the document without retaining anything in memory beyond the potential side effects of the event handlers.

2. The Document Object Model (DOM)

The XML DOM is a specification published by the W3C that is currently on its third version, designated DOM Level 3 (W3C, 04-4). Conceptually, DOM is fairly simple. As a DOM parser traverses a document, it builds a content tree consisting of various types of nodes along the lines of the example shown in Figure 3.3. The simple document of the example specifies a single UUV waypoint as a two-dimensional Cartesian coordinate and a commanded transit speed. The document uses attributes to specify the waypoint command's units of measure and element values to specify the command parameters. The document also contains a single comment. The resultant DOM tree contains a single Document node (at the root), Element nodes for each document element, Text nodes for element values, Attribute nodes for each attribute, and a Comment node for the example document's lone comment. It is worth noting that the value of a DOM element is maintained in a distinct object that is linked to the element, but is not a part of the element itself. Also, attributes are associated with the elements to which they apply, but they are not actually part of the tree (i.e., they do not have a parent node and are not allowed to have child nodes). Finally, comments are parsed and maintained in the parse tree along with the rest of the document.

In reality XML DOM has significantly broader applicability than XML parsing—it is actually a robust API for processing XML documents. In the DOM API, all nodes inherit from a common base node type and therefore have common characteristics and methods. Additionally, each node type has characteristics and methods specific to its subtype. Since the entire DOM tree is loaded into memory when a document is parsed, the tree can be manipulated programmatically using the methods associated with the

various nodes. New nodes can be created and added to the tree and existing nodes can be modified or deleted. In fact, the entire DOM tree can be generated programmatically rather than loaded from an existing file. The ability to programmatically generate and manipulate XML documents is perhaps DOM's most significant asset.

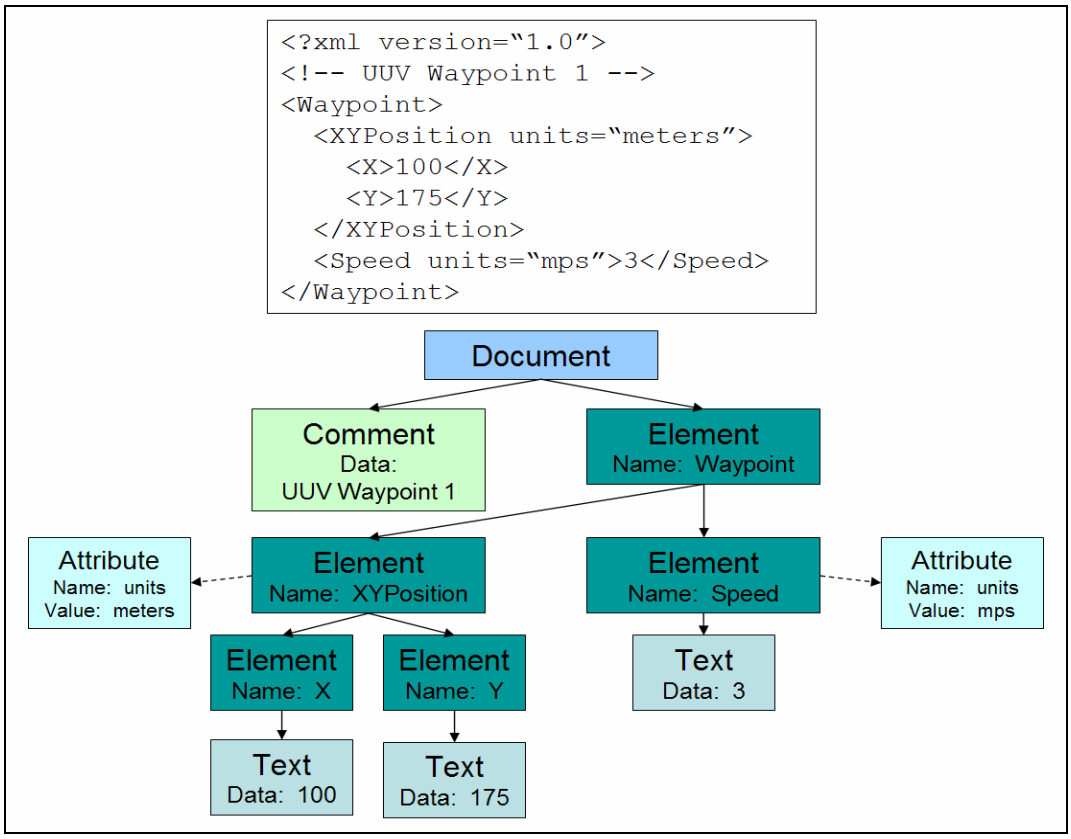


Figure 3.3. A Graphical Depiction of an XML Document Object Model (DOM) Tree Corresponding to a Simple XML Document Specifying a UUV Waypoint

The most significant disadvantages of DOM stem from the requirement to maintain the entire document in memory. If documents are small, the associated overhead is usually acceptable. However if documents are large, DOM can impose unacceptable memory and time requirements, particularly if only a small amount of information is to be extracted from a document or if multiple documents are to be loaded (Hunter, et al., 04)

3. The Simple Application Programmer's Interface (API) for XML Parsing (SAX)

The most commonly utilized alternative to DOM is SAX. Originally published as a Java API (but now available in a number of high-level programming languages), SAX

was not developed by and is not owned by any consortium, standards body or company. Nevertheless, SAX has evolved into a de facto standard upon which numerous applications rely (Hunter, et al., 04).

SAX is an event-driven parser. As the parser traverses a document, events (partially listed in Table 3.2) are triggered as certain constructs are encountered. The most common event types are those that are triggered when the parser encounters the document beginning or end, element beginning or end tags, character data (i.e., the element value) and whitespace. The parent application is responsible for reacting to events based on the type of event that was triggered and its associated parameters (e.g., the element name and attribute values associated with a startElement event). Thus the application is free to process events that are relevant and ignore ones that are not.

Event	Description
StartDocument	Event to notify the application that the parser has read the start of the document.
EndDocument	Event to notify the application that the parser has read the end of the document.
StartElement	Event to notify the application that the parser has read an element start tag.
EndElement	Event to notify the application that the parser has read an element end tag (will be fired immediately after the startElement event for empty elements).
Characters	Event to notify the application that the parser has read a block of characters.
IgnorableWhiteSpace	Event to notify the application that the parser has read a block of whitespace that can probably be ignored.
ProcessingInstruction	Event to notify the application that the parser has read a processing instruction.
StartPrefixMapping	Event to notify the application that the parser has read an XML namespace declaration and that a new namespace prefix is in scope.
EndPrefixMapping	Event to notify the application that a namespace prefix is no longer in scope.

Table 3.2. A Subset of Available Event Types that are Triggered during Simple Application Programmer's Interface (API) for XML (SAX) Parsing (After: Hunter, et al., 04)

Unlike DOM, once the parser completes its traversal, no document information is maintained beyond any side effects of event processing on the part of the parent application. For this reason, SAX's memory requirements are significantly reduced in

comparison to those of DOM. Further, since the parser does not need to develop a potentially large content map as it traverses the document, SAX can be significantly faster than DOM (Means and Bodie, 02). The main disadvantage to SAX is that it is only a parser and cannot be used to create or manipulate XML data.

Neither SAX nor DOM, therefore, is universally applicable. Each has advantages and disadvantages that are more suitable for some applications and less suitable for others. In general, SAX is preferable for processing XML streams or documents that can be discarded once processed while DOM is preferable for processing XML documents that are to be modified and for programmatically generating XML content.

E. XML DATA BINDING

SAX and DOM are usually acceptable mechanisms for parsing, generating and manipulating fairly simple XML documents. In fact they provide the only well-known API options for XML documents whose content is not constrained by a DTD or schema. However, for documents with well-defined but complex content models, such as the common autonomous vehicle data model proposed here, their use is cumbersome and error prone. The generic nature of DOM and SAX mean that the application developer must keep the entire content model in mind and explicitly account for every possibility when writing software to process XML. Further, neither SAX nor DOM has support for the data types with which developers are familiar. All attribute and element values in SAX and DOM are strings, and their numerical or Boolean values must be parsed from the string accordingly. Finally, SAX and DOM do not have the capability to detect even simple programming mistakes (e.g., a misspelled element name). In many cases a runtime exception will be thrown (often in a completely different portion of the application than the actual mistake), but it is equally likely that the application will simply not work correctly in some instances. This lack of error detection in SAX and DOM can make troubleshooting extremely difficult.

Fortunately, a mechanism exists to more reliably and efficiently develop and process XML documents that are governed by a schema or DTD, namely XML data binding. Stated simply, XML data binding is the use of a schema or DTD to automatically generate a customized API for the manipulation of compliant XML documents (McLaughlin, 01). Schema-specific APIs allow the developer to work with

elements and attributes by name and data values by type. They also provide well-named get and set accessor methods for the named elements and complex types. Naming and typing conventions allow the compiler to detect many common errors (e.g., incorrect data typing, misspellings, etc.) when the application is compiled, making them much easier to trouble shoot. Finally, schema-specific APIs can preclude the generation of invalid documents by enforcing compliance while the programming objects corresponding to a document are being constructed, manipulated in memory, or written out.

As with other types of XML utilities, there are numerous data binders available that produce APIs in a variety of programming languages. Output language and style differences aside, the end products of most XML data-binding utilities are similar. Application of a data-binding utility to a schema or DTD typically results in the generation of a marshaller, an unmarshaller, a validator and a content-specific API (McLaughlin, 02). The marshaller and unmarshaller are respectively used to generate data-bound programming objects corresponding to XML documents and to serialize data-bound objects back to XML. The validator is used to check the validity of data-bound objects against the schema. Finally, the content-specific API contains all of the methods required to access and manipulate data-bound objects. Additionally, the content-specific API can be used to generate data-bound objects from scratch.

In practice, XML data binding is used as graphically depicted in Figure 3.4. The XML data binder is applied to the schema to generate the marshaller, unmarshaller, validator and content-specific API. A client application uses these products to load valid XML documents into data-bound objects or generate them from scratch; access, manipulate and validate data-bound objects; and write data-bound objects out as XML. The application, therefore, never deals directly with the XML documents and manipulates data-bound objects using the content-specific API. Additionally, neither the application nor the data binder products require runtime access to the schema or data binding utility. This has the advantage of enabling vehicle applications and remote planners that may not have network access to the master schema to enforce document validity offline.

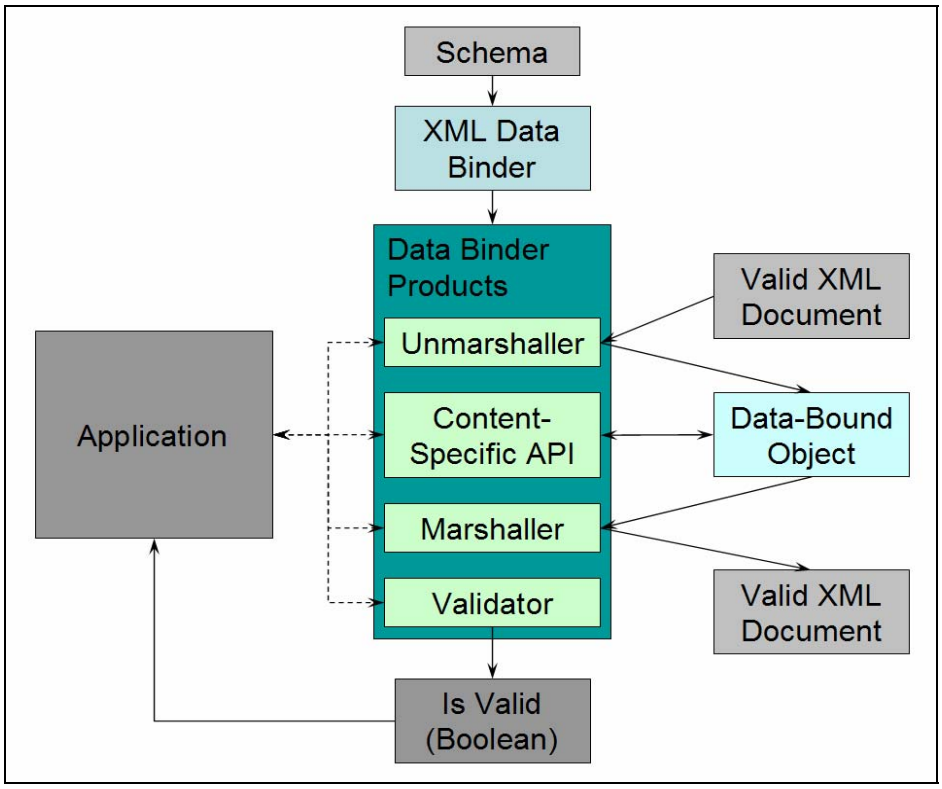


Figure 3.4. A Graphical Depiction of the Interactions Between an XML Data Binding Utility, the XML Schema, the Binder Products, XML Documents and a Client Application

XML data binders use common-sense heuristics to map XML Schema or DTD characteristics to classes in the output programming language. These heuristics provide for consistent access to document elements and attributes. A summary of some mapping heuristics of the data binding utility used in this research, Sun Microsystems' Java Architecture for XML Binding (JAXB), are provided in Tables 3.3 through 3.5.

Typical of most binders, JAXB maps XML Schema simple types to the closest matching available type of the output programming language (in this case Java). In cases where the schema and output language types are not perfectly aligned, such as the JAXB mappings of XML unsigned integer types, the validator (and possibly the marshaller and unmarshaller) are typically be responsible for ensuring assigned value compliance with the schema. Complex XML types normally map to classes or interfaces in the output language. Naming of the produced classes generally correspond to the name of the element or complex type to which they correspond. Similarly, accessor methods are

typically generated with names that correspond to the name of the attribute or element that they address. In cases where an element can have multiple children of the same element type, a single accessor is normally generated that returns a linked list containing all of the relevant child elements. List-manipulation methods of the output language are then used to access and manipulate the individual elements. In programming languages like Java, where the contents of a linked list can be of any type, the validator is responsible for ensuring all elements in the list are of the appropriate type. In other languages, however, the data-binding heuristics may be able to preclude list elements of the wrong type.

Simple XML Types	Java Type or Class Mapping
byte	Maps to byte.
short	Maps to short.
int	Maps to int.
unsignedByte	Maps to short. Range validation is the responsibility of the validator.
unsignedShort	Maps to int. Range validation is the responsibility of the validator.
unsignedInt	Maps to long. Range validation is the responsibility of the validator.
integer	Maps to BigInteger.
float	Maps to float.
double	Maps to double.
decimal	Maps to BigDecimal.
string	Maps to String.
Boolean	Maps to boolean.
Value-restricted simple type	Maps to same Java primitive data type or class as the parent simple type. Value validation is the responsibility of the validator.

Table 3.3. Java Architecture for XML Binding (JAXB) Heuristics for Mapping XML Schema Simple Types to Java Types (After: Sun, 05)

Complex XML Types	Java Mapping
Named element (empty)	JAXB-defined AnyType class.
Named element (simple content)	Maps to the Java type of the simple content.
Named element (complex content)	JAXB-defined class with the same name as the element tag.
Named complex type	JAXB-defined class of the same name as the XML complex type.
Complex content embedded in a parent type or element	JAXB-defined subclass within the parent element or type class (e.g., ParentClass.SubClass).
Named group	No type mapping, handled with accessor method mappings.

Table 3.4. JAXB Heuristics for Mapping XML Schema Complex Types to Java Classes (After: Sun, 05)

Content Type	Accessor Description
Element attributes (user-defined value)	Get and set methods with the attribute name (e.g., setAttribute and getAttribute) and a parameter or return value of the attribute type.
Element attributes (fixed value)	Class variable with the attribute name (e.g. VARIABLE) and constant value.
Element value (text or numerical)	Get and set methods with the element name (e.g., setElement and getElement) and parameter or return values of the element type.
Child elements specified by name (single instance)	Get and set methods with the child element name (e.g., getElement and setElement) and a parameter or return value of the child element type.
Child elements specified by name (multiple instance)	Get method with a return value of the Java List type. List methods (e.g., get, add, remove, etc.) are used to access and modify elements. The validator is responsible for ensuring the List contains only the appropriate elements.
Child elements specified by group (single instance)	Get and set methods for each allowable element as defined by the group. The validator is responsible for ensuring the content complies with the group definition (e.g., choice, all, sequence, etc.).
Child elements specified by group (multiple instance)	Get method with a return value of the Java List type. List methods are used to access and modify elements. The validator is responsible for ensuring the List contains only appropriate elements and that the content complies with the group definition.

Table 3.5. JAXB Heuristics for Schema-Governed XML Element and Attribute Accessors (After: Sun, 05)

XML data binding is a powerful tool in the development of applications that process DTD or schema-governed XML data. As such it is utilized extensively in this research for the implementation of mission planning systems and vehicle controllers as well as in the processing of various configuration files. Additionally, it is an integral part of the translation mechanism for converting between data-model-compliant XML and existing autonomous vehicle data formats (both bit-mapped binary and non-XML text). The role of XML data binding in these conversions is discussed in detail in Chapter V.

F. EXTENSIBLE STYLESHEET LANGUAGE FOR TRANSFORMATION (XSLT)

Among the most useful tools available to XML applications is XSLT. Specifically defined for the purpose transforming one XML document into another (W3C, 01), XSLT is by no means limited to this application. As demonstrated in (Neushul, 03), it can be used effectively to convert XML documents into virtually any text-based format containing the same information, or derivable information, as the original XML document. Common uses for XSLT now include the generation of

documentation, graphics files (X3D, Scalable Vector Graphics, Virtual Reality Modeling Language, etc.) and even program source code. Among web-based applications, XSLT has evolved into the *de facto* standard for the transfer of data between applications requiring different formats (Kay, 03), a process that is semantically identical to transforming an XML document constrained by an autonomous vehicle data model to a vehicle-specific format.

Although XSLT has been proven to be Turing complete (Holman, 02), it has a number of characteristics that make it somewhat atypical among programming languages. These are clearly illustrated by a quick examination of a few of XSLT's design goals (Tidwell, 01):

- An XSLT stylesheet is itself an XML document. This means that it can be processed by a variety of XML utilities and can even be transformed by another XSLT stylesheet.
- XSLT is a pattern-matching language wherein templates define the desired output for matching source document constructs. XSLT programs are declarative in style, as opposed to the imperative nature of Java, C, or C++ programs.
- XSLT is designed to be free of side effects meaning that the execution of one template will not affect the execution of subsequent templates. The most significant implication of this is that all variables are immutable—once declared, the value cannot change.
- XSLT has only two branching constructs: if-then and choose-when-otherwise. Neither construct is capable of inadvertently invoking an infinite loop.
- XSLT has no looping constructs. Rather, it relies on iteration and recursion to accomplish repetitive tasks.

From a mechanical standpoint, XSLT is actually composed of two specific components. The first is the XSLT language itself which is comprised of 37 elements that provide all of XSLT's functionality. The second integral part of XSLT is the XML Path Language (XPath), an expression language that is used to define criteria for template matching and selecting nodes and values in an XML document. XPath consists of functions that perform operations on or extract information about nodes, strings and numbers; arithmetic and logical operators; a mechanism for specifying a search axis (e.g., descendants, siblings, or ancestors of the current node); and a selection mechanism for specifying which nodes, attributes and values meet the criteria of an expression.

XPath expressions are found in XSLT stylesheets as attribute values of elements pertaining to template matching or expression evaluation.

The execution of an XSLT stylesheet progresses sequentially until it ends or more commonly reaches an "apply-templates," "call-templates" or "for-each" element. At this point it evaluates any associated XPath expressions and perform the required actions on each selection of the expression. Templates can be called by name but are more commonly invoked when the current node matches the template's XPath criteria. In instances where more than one template matches the current node, only the template corresponding to the most specific match is executed. Templates can be invoked from within other templates with program flow ultimately dependent as much on the structure and content of the source document as on the XSLT stylesheet.

Not surprisingly, the template-matching pattern of XSLT is tailor made for the transformation of XML documents. XSLT's use of immutable variables raises difficulties when it is used to transform AVCL documents because of a requirement to maintain up to date state information throughout the transformation. However, this difficulty was overcome through the development of an XSLT pattern that uses template parameters to mimic the functionality of mutable variables. In general, XSLT provides a powerful mechanism for converting data-model XML documents to various vehicle-specific formats and receives significant attention in Chapter V.

G. BINARY XML AND XML COMPRESSION

While the preceding discussion focused primarily on aspects of XML that support its use in a common autonomous vehicle data model, XML documents have one characteristic that provides a strong counter argument. As a rule the use of XML results in documents that are significantly larger than non-XML representations of the same data. Using AVCL for UUV mission results files, for instance, results in files that are roughly two and one half times larger than space-delimited text files containing the same data. When compared to binary data formats, the size increase is even more striking. An XML-encoding of 21 common JAUS messages, for instance, results in up to a 60-fold increase in size over the standard binary encoding. This document-size disparity is not surprising since terseness was specifically excluded as an XML design consideration. In

the autonomous vehicle domain, the size of XML documents can impose potentially prohibitive bandwidth and processing requirements.

In an effort to address the shortcomings of XML for certain applications, most notably the processing, memory, and bandwidth requirements imposed by XML's intentional lack of terseness, the W3C recently established the XML Binary Characterization Working Group. The goal of this working group is to explore the feasibility and applicability of a binary XML format and ultimately to develop a binary XML standard (W3C, 05). Binary XML is "a format which does not conform to the XML specification yet maintains a well-defined, useful relationship with XML" (W3C, 05). Stated another way, a binary XML document is logically equivalent to an XML document but does not comply with the XML specification. A key point is that it is possible to unambiguously and reversibly convert between standard XML and binary XML, preserving all relevant information.

The XML Binary Characterization Working Group efforts focused on determining use cases and defining requirements for binary XML rather than the development or endorsement of a particular standard. The proposed requirements include a list of characteristics that binary XML must support and a list of characteristics that binary XML must not prevent. The "must support" list consists of characteristics such as platform neutrality, streamability and transport independence that are considered essential to the success of binary XML (i.e., binary XML will be unable to meet the requirements of the anticipated use cases if it does not exhibit these characteristics). The "must not prevent" list includes characteristics such as processing efficiency, implementation cost and forward compatibility that relate to binary XML's utilization and growth.

The task of actually defining a standard binary XML encoding has fallen to the Efficient XML Exchange Working Group, a follow-on to the XML Binary Characterization Working Group. A number of binary XML formats are currently under development with encoding strategies generally falling into one of two categories—schema-based and non-schema-based. Schema-based encodings use information contained in the schema to encode a document. Non-schema-based encodings, on the other hand, rely solely on the content and structure of the document being encoded.

Regardless of whether a binary XML format is schema-based or non-schema-based, the basic underpinnings normally involve the replacement of XML character sequences with more efficient data types. At the center of both Fast Infoset (a non-schema-based encoding method) (ITU, 05) and XSBC (a schema-based technique) (Serin, 03), for instance, are a set of lookup tables that are used to replace tag and attribute names, namespace prefixes and other common XML document constructs with integer references. Both also replace character-based representations of numerical data with more efficient representations such as the Institute of Electrical and Electronics Engineers (IEEE) integer and floating point formats. Finally, both eliminate superfluous content such as white space, end tags, quotation marks and other XML formatting characters (e.g., '<' and '>').

Among the differences between schema-based and non-schema-based binary XML formats are the method by which the lookup tables are generated and whether or not the lookup tables are part of the binary XML document. Fast Infoset, typical of non-schema-based techniques, can generate the lookup tables as a document is encoded and most Fast Infoset documents include the tables at the beginning of the binary XML document. Fast Infoset also allows the use of a reference to external lookup tables as an alternative to including the tables in the document itself. This option is desirable for encoding small documents since the lookup tables in these cases are large relative to the rest of the encoding.

Schema-based encoding techniques provide the option of building lookup tables before the document is parsed and saving them for future use rather than regenerating them for each document that is encoded. Since lookup tables for schema-dependent encodings are independent of the documents themselves (i.e., the lookup tables for all documents complying with a given schema will be identical), the tables themselves do not need to be included in the binary XML document but can be generated by the application when the document is to be decoded.

Since schema-based encoded binary XML documents do not directly incorporate the lookup tables, they have the advantage of smaller size (although this advantage disappears if the non-schema-based encoding uses a reference to a set of external lookup

tables). This is particularly evident in the smaller documents of Figure 3.5, which compares Fast Infoset and XSBC encodings of binary JAUS messages encoded as XML. Relatively small binary messages (typically in the neighborhood of 20 bytes), the XML encodings impose anywhere from a 25- to 60-fold increase in size (37.82 average). Fast Infoset encoding of the XML results in only a small reduction in size, still 20 to 50 times the size of the equivalent binary messages (30.31 average). Application of further compression to the Fast Infoset documents results in only minimal improvement (still 22.18 average times larger than binary). The same documents encoded with XSBC show significantly better results—from three to six times the size of the original binary messages (4.63 average). Additionally, further compression of the XSBC-encoded documents in this case produces only negligible improvement (4.57 average times larger). Although these documents are still significantly larger than the original binary messages, they are not so large as to preclude transmission over circuits typically used by autonomous vehicles.

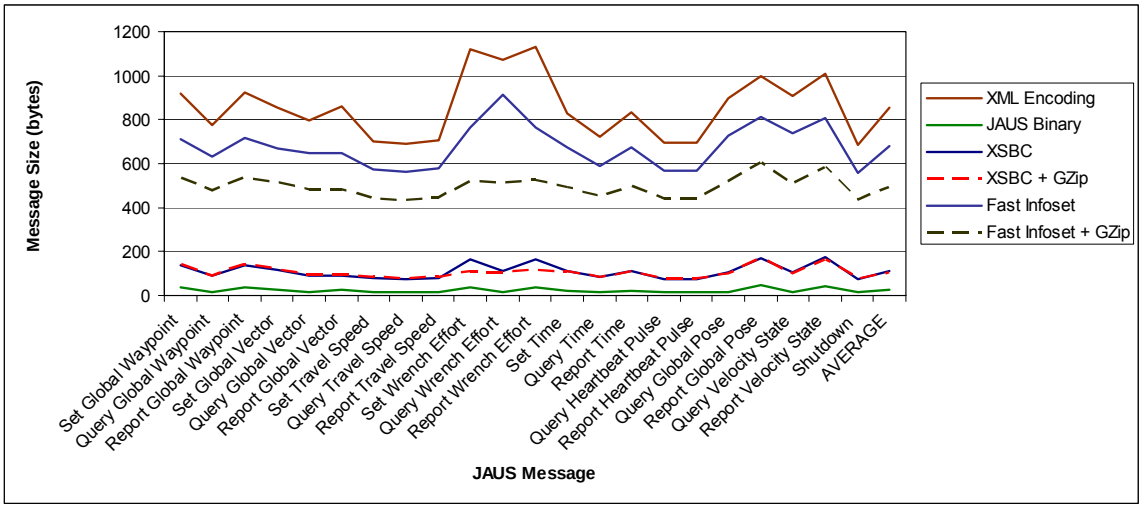


Figure 3.5. A Comparison of Fast Infoset and XML Schema-Based Binary Compression (XSBC) of XML Encoded Joint JAUS Messages to the Standard Binary Encodings and Uncompressed XML

As evidenced by Figure 3.6, the size advantage of schema-based encoding techniques disappears with larger documents. This figure compares Fast Infoset and XSBC encodings of large AVCL documents (7 to 30 megabytes in size). For comparison purposes, the results are compared against space-delimited text documents containing

identical data. The AVCL documents are roughly two and one half times larger than the space-delimited text documents, but application of either Fast Infoset or XSBC results in documents that are smaller than the space-delimited text documents. Fast Infoset provided slightly better compression with an average document size of 61 percent of the space-delimited text (individual documents size ranged from 57 to 67 percent). XSBC encoded document size averaged 72 percent of the equivalent space-delimited text (with individual documents ranging from 66 to 76 percent). Further compression of either Fast Infoset or XSBC results using GZip resulted in an average size of 14 percent of the original space-delimited text (with individual documents ranging from 11 to 18 percent). Interestingly, applying GZip to the space-delimited text resulted in documents that were over twice as large as the GZipped binary XML.

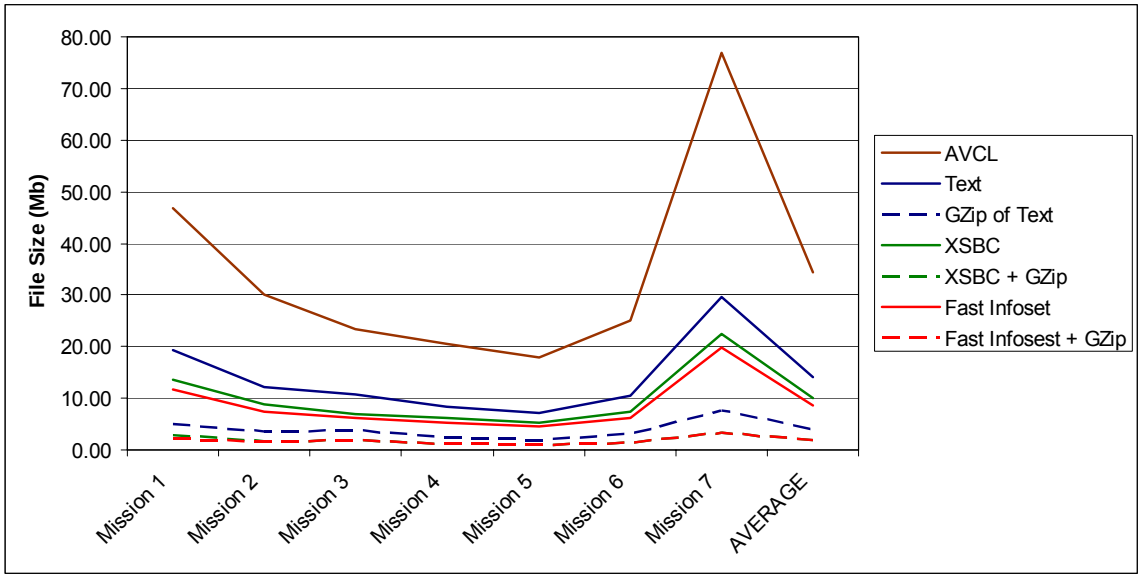


Figure 3.6. A Comparison of Fast Infoset and XSBC Compression of Autonomous Vehicle Command Language (AVCL) Mission Results Files

The preceding data points illustrate that binary XML has the potential to extend the use of XML into domains for which it was previously unsuited. A rapidly evolving field, the various methods for implementing binary XML have enough in common that in all likelihood, they will coalesce into a universal standard sooner rather than later. Until then, if binary XML is desirable, the requirements of the application can dictate which encoding method to use, in particular whether to use a schema-based or non-schema-based approach. For instance, if document size is an important factor and small

documents are to be encoded, it is probably advisable to use schema-based encoding (or to use a non-schema-based strategy that allows an external reference to the lookup tables). If only large documents are to be encoded, the decision is less critical, but non-schema-specific encoding might provide a performance advantage since these strategies do not parse an external schema. Obviously, if documents that are not schema-governed are to be encoded, non-schema-based encoding is warranted (alternatively, a synthetic schema can be generated based on the document content and structure).

H. SUMMARY

XML has a number of characteristics that make it potentially attractive for developing a common autonomous vehicle data model. Among the most important XML capabilities are the ability to rigorously constrain document structure and content through the use of XML Schema and the related ability to automatically validate instance documents as they are loaded, created or modified. Additionally, various standards are readily available to facilitate the application of XML to the domain of interest. XSLT and XML data binding in particular play an important role in the implementation of a common autonomous vehicle data model. Finally, emerging binary XML capabilities offer the promise of mitigating the one significant drawback to the use of XML in this area—its size.

When viewed as a whole, the APIs, software packages, and utilities available to the XML application programmer facilitate the development of easy-to-use applications for the generating and processing XML documents of a specific type. These applications are easily designed to validate document content and conduct error checking without requiring the end user to use or even understand the underlying XML structure. Considering that the potential users of autonomous vehicle systems are unlikely to possess significant XML expertise, this is an important capability for any vehicle support system that is intended for operational use. Thus, a compelling case can be made for the use of an XML-based autonomous vehicle data model even before the ability to translate content to other forms using XSLT is considered.

XML applications are most commonly written using the Java programming language. The platform neutrality of Java mirrors that of XML, so it was only natural to include robust XML processing in Java. Further, many of the XML utilities, standards

and applications that are utilized to support this research were designed and implemented with Java in mind. Unfortunately, the platform neutrality of Java is not without cost. Since Java byte code runs on a Java Virtual Machine instead of a computer's native processor, there is some overhead that can make Java applications less efficient than those written in languages that compile directly to machine-executable code.

A number of parties are working on "real-time Java" as well as hardware and software solutions to improve the efficiency of Java implementations. In time these efforts may lead to increased acceptance of Java as an appropriate programming language for real-time applications. For the time being however, developers still gravitate towards the use of programming languages such as C and C++ to implement control software. Fortunately, this does not stand as a barrier to the use of XML for an autonomous vehicle data model. As the advantages of XML have become more obvious and XML has been applied to a broader array of applications, the ability of languages other than Java to process XML has improved significantly. In fact implementations for all of the utilities and standards discussed in this chapter are available in a number of languages. Object-oriented languages such as C++ in particular are quickly becoming XML-capable (Arcineas, 02). Implementation languages, therefore, can be chosen on their own merits without inhibiting the use of XML as described here.

IV. AUTONOMOUS VEHICLE COMMON DATA MODEL DEVELOPMENT

A. DATA MODELS VERSUS ONTOLOGIES

Thus far the terms "data model" and "ontology" have been used more or less interchangeably. Researchers in the semantic web, knowledge management and ontology engineering fields will contend, however, that the terms imply similar but not quite interchangeable meaning. Nevertheless, in practice, the distinction between what constitutes a data model and what constitutes an ontology is nebulous at best. It is worthwhile, therefore, to briefly compare and contrast the salient characteristics of data models and ontologies in order to more precisely define what is being proposed and implemented by this research.

A data model describes in an abstract way how data is represented in a business organization, information system or database management system. A data model will rigorously characterize the data structure and content of the domain that it conceptualizes. Even so, the degree to which a model encapsulates semantics will depend on the expressiveness of the modeling formalism and the design decisions of the modeler. Generally speaking, the concept of a data model is significantly broader than that of an ontology in that many ontologies might also be considered data models, but few data models can also be considered ontologies.

Although there exist many computer ontology definitions, they tend to include a number of common elements. It is generally agreed that an ontology is an "agreement about a shared, formal, explicit and partial account of a conceptualization" (Guarino and Giaretta, 95). At its core, an ontology contains the vocabulary, concept definitions and relationships for a given domain. Part and parcel to any ontology, therefore, is the definition of domain rules that restrict and characterize the semantics of concepts and relationships. Further, for a data model to qualify as an ontology it must be machine interpretable—that is, an application using the model must be able to infer the semantics without *a priori* knowledge of the model (Daconta, et al., 03). Finally, it is assumed that applications "commit to" the semantics of an ontology prior to using it (Spyns, et al., 02).

Although these definitions provide some qualitative insight into the difference between a data model and an ontology, they do not effectively differentiate between the two or definitively classify the exemplar developed in the course of this research as one or the other. This common quandary has led a number of researchers in the field to use rather arbitrary criteria to make the distinction. Some differentiate based on the intended application domain—that is, data models apply to a single application or a single application type while ontologies apply to any application that is interested in the ontology's domain (Gottgtroy, et al., 03)(Kalinichenko, et al., 03)(Jarrar, et al., 03). Others might make the distinction based on the modeling formalism utilized. Entity-Relationship and Extended Entity Relationship diagrams (Ramakrishnan and Gehrke, 03), Object Role Modeling (Halpin, 01), Unified Modeling Language (OMG, 05) and XML Schema, for instance are used to develop data models, while the Defense Advanced Project Agency (DARPA) Agent Modeling Language + Ontology Inference Layer (DAML+OIL) (DARPA and IST, 01) and Web Ontology Language (OWL) (W3C, 04-5) are used to develop ontologies.

Similar, but somewhat more subjective than using the intended application domain as a classification criterion is the notion of "genericity." This method of classification assumes that in order to be sharable and usable by a broad array of applications, ontologies must be more generic than data models which are not intended to be universally sharable. (Spyns, et al., 02) proposes four characteristics relating to genericity that can be examined to determine whether a particular model should be considered an ontology. First is the operational level of the intended data that the model's rules constrain—do the rules apply at the implementation level (data types, ranges, keys, etc.) or are they more abstract. The concept of expressive power refers to the data-engineering language utilized by the model—the degree to which it is concerned with defining structure and data integrity versus the ability to express meaningful constraints and relationships. The third proposed measure of genericity is user purpose and goal relatedness, a subjective assessment of the how much influence the intended use of the model has on its design. Finally, the extendibility of the model (i.e., the ability to add to or modify aspects of the model without impacting unrelated portions of the model) is considered. It is worth noting that the authors of this paper are not proponents of large

monolithic ontologies defined with languages such as DAML+OIL or OWL, but prefer the layered, highly compartmentalized approach described in the paper. Not surprisingly, this method of ontology / data model differentiation tends to favor the authors' preferred method of ontology definition over others.

A final method of data-model classification that avoids these somewhat arbitrary distinctions between data model and ontology utilizes a subjective scale along the lines of the ontological spectrum of Figure 4.1 (Daconta, et al., 02). Here the goal is not specifically to determine whether or not a data model can be considered an ontology, but rather to assess the level of semantic richness of the model. All models classifiable on the ontology spectrum can be considered ontologies to a degree, however models with stronger semantics are more accurately classified as ontologies in the generally accepted sense than those with weaker semantics.

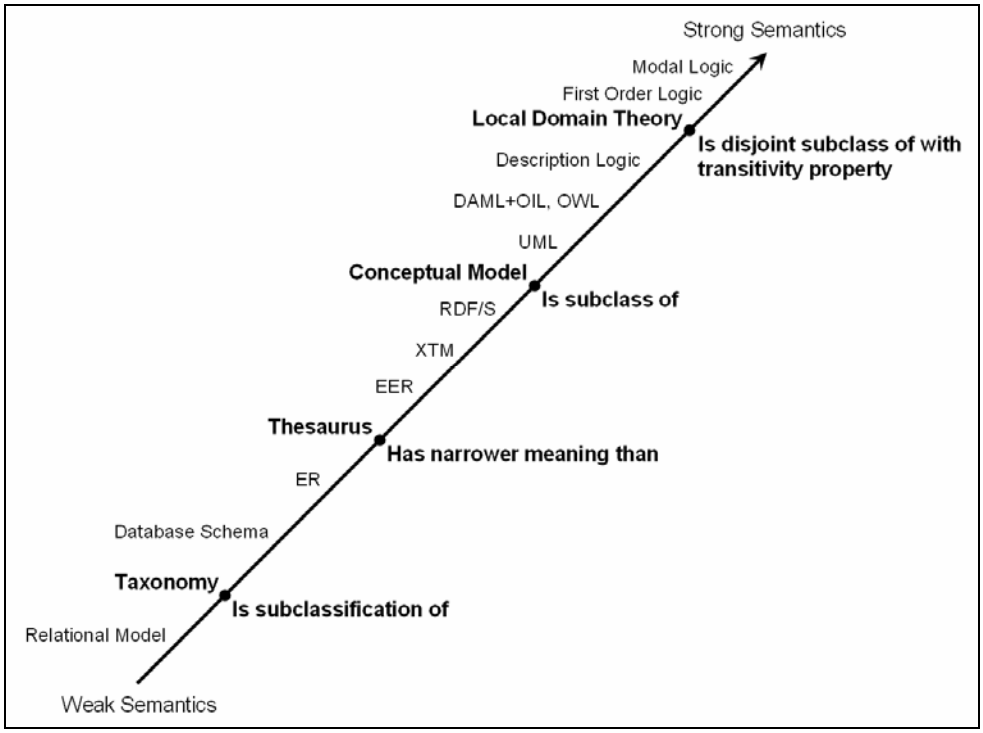


Figure 4.1. The Ontology Spectrum (Weak to Strong Semantics) Demonstrating the Relationship between Ontology Types and Expressive Power (From: Daconta, et al., 02)

The right side of the ontology spectrum of Figure 4.1 provides examples of the types of relationships that can be effectively expressed by a model locatable on that

portion of the scale with associated named model types (described in detail in (Daconta, et al., 02)) on the left (in bold). The modeling languages on the left side of the scale are not used to classify models but rather to indicate the semantic strength that the respective languages are capable of expressing (e.g., a model expressed with DAML+OIL can be semantically weaker than a model expressed using an Extended Entity Relationship diagram despite the fact that DAML+OIL is significantly more expressive). Specific models, therefore must be classified on their own merits as opposed to the merits of the language in which they are defined.

The preceding discussion makes it clear that the distinction between a data model and an ontology is somewhat subjective. It does, however, provide criteria by which to evaluate the type of data model that is proposed by this research. Based on the arbitrary criteria of application domain (autonomous vehicle command and control) and model definition language (primarily XML Schema), AVCL does not qualify as an ontology. Similarly, based on the four suggested measures of genericity, AVCL is more accurately categorized as a data model than an ontology—its rules constrain implementation-level data, XML Schema constrains data content and structure but does not explicitly address semantics, the intended use of the model is the primary influence on its design, and the extendibility of the model can be cumbersome since changes have the potential to invalidate data instances that were previously compliant. Finally, AVCL is probably best placed on the lower half of the ontology spectrum of Figure 4.1 due to the inability of XML Schema to explicitly express the strong semantics required by the upper portion of the spectrum in a machine interpretable manner. It is more accurate then to describe the type of data model proposed by this research (with AVCL serving as the exemplar) as an XML-Schema governed data model in the domain of autonomous vehicle command and control than as an ontology in the same domain.

B. AUTONOMOUS VEHICLE DATA MODEL DEVELOPMENT

1. Overview

The development of a data model of the type proposed by this research will have two primary influences. First are the research goals described in Chapter I. A suitable data model must contain a declarative, goal-based mission specification capability and associated goal set, a script-based, task-level mission specification capability and

associated task-level behavior set as well as an intervehicle messaging capability and associated message set. The second driver of data model design is vehicle capability. The task-level behavior and message sets must be compatible with the actual capabilities of the target vehicles. Stated differently, it must be possible to express target vehicle tasking and messaging using the data model's task-level behavior and messaging constructs respectively. Also, while not specifically required by the research goals, the exemplar data model of this research incorporates a mission results sections to facilitate post-mission data analysis. Since the data model is intended to encapsulate implementation-level data, it requires well-defined data types, units and conventions at a more basic level in order to achieve the granularity necessary to operate with actual vehicles. The remainder of this chapter describes the implementation of these data model attributes in the development of AVCL in order to illustrate design issues inherent in the development of a data model of this sort.

2. Data Types and Conventions

a. Units and Conventions

The various command languages and data formats designed for real-world vehicles normally use specific units of measure and explicitly defined conventions to express and communicate vehicle characteristics and state parameters such as position and posture. Since these vehicle-specific languages were designed around their respective target vehicles, they typically do not have cross-vehicle compatibility in mind. Therefore, the units of measure and other conventions often differ significantly from one vehicle to another. Fortunately, conversion between the units of measure of different vehicle-specific data formats requires only multiplication by a scaling factor (units of measure utilized in AVCL are shown in Table 4.1). Reconciliation of most other vehicle-specific data format conventions, while somewhat more complicated, does not pose any great difficulty (Table 4.2 provides a summary of the conventions utilized in AVCL).

Among the more difficult conventions to resolve are the various methods for expressing vehicle position and orientation. Available methods for specifying a position (location on the face of the earth), for instance include Cartesian coordinates in a fixed reference frame, latitude and longitude, Military Grid Reference System and Universal Transverse Mercator grid, among others. Conversion between these

conventions can be tedious if a high degree of precision is required, but various Geographical Information Systems and algorithms are available for this purpose. The decision of which conventions to allow in the data model, therefore, can be left up to the modeler without impacting the model's applicability to any particular vehicle-specific format so long as the Geographical Information System is utilized in the conversion process between the vehicle-specific and common data models.

Measurement	Unit	Note
Linear Distance	meters	
Angular Distance	degrees	Used for heading, course, bearing, etc. but not control surface deflection.
Linear Speed	meters per second	
Linear Speed	knots	Nautical miles per hour, used as a convenient format for commanded speed.
Angular Speed	degrees per second	
Time	seconds	Can be relative to a fixed start time or current time depending on context.

Table 4.1. Standard Units of Measure used throughout the AVCL Schema

In the case of AVCL the decision was made to allow positions to be specified with Cartesian coordinates or latitude and longitude. Ultimately, positions (including the vertical component) are placed in a right-handed, earth-fixed reference frame with the positive X axis directed north, the positive Y axis directed east and the positive Z axis directed down. Conversions between AVCL and vehicle-specific data formats that use other coordinate systems require multiplication by a rotation matrix as part of the conversion process. Other than this simple conversion prerequisite, the decision to use this particular coordinate system carries no overhead (McGhee, et al., 00).

There are also a number of methods commonly used to specify orientation including Euler angles, gimble angles, rotation matrices, angle / axis pairs, and quaternions. Each of these methods has advantages and disadvantages, but as with position, translation of an orientation specification from one format to another is not difficult. A detailed discussion of many of the issues arising from the use of competing coordinate system and orientation conventions can be found in (McGhee, et al., 00). Therefore, the decision to use Euler angles in AVCL (based on the assumption that users will find bank, pitch, and yaw more intuitive than other methods) does not inhibit the

compatibility of AVCL with any vehicle-specific method for specifying orientation. Additionally, the use of Euler angles within AVCL does not preclude the use of another method for encoding orientation at the application level.

Convention	Description
Earth-Fixed Coordinate System	Right handed three-dimensional system with the positive X axis directed north, positive Y axis directed east and positive Z axis directed down.
Body (Vehicle)-Fixed Coordinate System	Right handed three-dimensional system with positive X axis forward, positive Y axis to the right, and positive Z axis out the vehicle bottom.
Horizontal Position	Latitude / Longitude--signed degrees and decimal degrees.
Horizontal Position	Cartesian Coordinates--(x, y) coordinate in the earth-fixed reference frame.
Horizontal Position	Relative Position--displacement from the current vehicle's position in earth-fixed coordinates (meters north and east of current vehicle position).
Orientation	Euler Angles (bank, pitch, and yaw or Φ, θ and Ψ)--in order rotation about the earth-fixed X, Y, and Z axes (or equivalently, in-order rotation about body-fixed Z, Y, and X axes).
Linear Velocity	Meters per second rate of travel relative to the earth-fixed coordinate frame $(\dot{x}, \dot{y}, \dot{z})$
Linear Velocity	Meters per second rate of travel relative to the body-fixed coordinate frame (u, v, w)
Angular Velocity	Degrees per second Euler angle rate $(\dot{\phi}, \dot{\theta}, \dot{\psi})$.
Angular Velocity	Degrees per second rate of rotation about the body fixed coordinate system (p, q, r).
UUV Vertical Position	Depth below the surface.
UUV Vertical Position	Altitude above the sea floor.
UAV Vertical Position	Altitude above mean sea level.
UAV Vertical Position	Altitude above ground level.
Actuator Setting	Percentage of maximum actuator authority (possibly signed).

Table 4.2. Miscellaneous Conventions used in AVCL

b. Simple Data Types

Specifications for non-XML, vehicle-specific data formats must often precisely define even simple data types. JAUS, for instance, mandates the use of the eight integer types and two floating point types of Table 2.2, and also defines a method for defining scaled integers where an integer data type is used as an index into a finite set of real numbers occurring at fixed intervals between a lower and upper bound (JAUS, 04-3). Additionally, the JAUS Reference Architecture must deal with byte ordering both of individual data units and data streams. An XML-based data model, on the other hand, makes much of the low-level data type definition unnecessary. Tables 4.3 and 4.4

contain descriptions of built-in primitive and derived data types respectively that can be used by any XML Schema-governed data model. Rigorous definition of these data types in the XML specification provides XML parsers, XSLT processors, XML data-binders, XML binary encoders and other applications all of the information necessary to process data instances. Further definition in the context of a particular data model is therefore not required.

Subtype	Primitive Data Type	Description
String Types	string	A finite-length sequence of characters.
	anyURI	A standard internet uniform resource identifier (URI).
	NOTATION	Declares links to non-XML content and associates it with an external application.
	QName	A namespace-qualified XML name.
Encoded Binary Types	Boolean	true or false.
	hexBinary	Binary data represented as a series of two-character hex strings.
	base64Binary	A binary encoding of a limited set of ASCII characters.
Numeric Types	decimal	A floating point number of arbitrary precision.
	float	An IEEE single-precision 32-bit floating point number.
	double	An IEEE double-precision 64-bit floating point number.
Date / Time Types	duration	Specifies a duration in years, months, days, hours, minutes and seconds.
	dateTime	Specifies a specific time of day on a specific Gregorian calendar date.
	date	Specifies a Gregorian calendar date.
	time	Specifies a time of day.
	gYearMonth	Specifies the year and month of a Gregorian calendar date.
	gYearDay	Specifies the year and day of a Gregorian calendar date.
	gYear	Specifies the year of a Gregorian calendar date.
	gMonthDay	Specifies the month and day of a Gregorian calendar date.
	gMonth	Specifies the month of a Gregorian calendar date.
	gDay	Specifies the day of a Gregorian calendar date.

Table 4.3. Predefined XML Schema Primitive Datatypes

The predefined data types available in XML Schema suffice for many elements of most data models, but they are often insufficient for at least some model

elements. The use of XML Schema, therefore, does not normally absolve the modeler from the responsibility of defining special simple data types to fit the requirements of the various model domains. For numerical and date data, schema-defined data types usually specify a continuous or discrete range of values. For string and encoded binary types, a finite set of potential values is normally defined (referred to as enumeration values).

Base Primitive Type	Derived Data Type	Description
string	normalizedString	A string with each white space character replaced with a space character.
	token	A string with leading and trailing white space eliminated and internal white space reduced to single space characters.
	language	A natural language two or three character identifier string.
	Name	A valid XML 1.0 element or attribute name.
	NCName	A valid XML 1.0 element or attribute name that prohibits the use of the ':' character.
	ID	A token that is used as a unique identifier for an element.
	IDREF	A reference to an element identified with an ID.
	IDREFS	A space-delimited sequence of IDREFs.
	NMTOKEN	Similar to NCName, but allows leading numbers.
	NMTOKENS	A space-delimited sequence of NMTOKENs.
	ENTITY	An NCName that refers to a pre or user-defined "entity" that is to be inlined.
	ENTITIES	A space-delimited sequence of ENTITYs.
decimal	integer	Any integer number.
	negativeInteger	Any integer number with a value less than zero.
	positiveInteger	Any integer number with a value greater than zero.
	nonNegativeInteger	Any integer number with a value greater than or equal to zero.
	nonPositiveInteger	Any integer number with a value less than or equal to zero.
	byte	An 8-bit signed integer.
	short	A 16-bit signed integer.
	int	A 32-bit signed integer.
	long	A 64-bit signed integer.
	unsignedByte	A non-negative 8-bit integer number.
	unsignedShort	A non-negative 16-bit integer number.
	unsignedInt	A non-negative 32-bit integer number.
	unsignedLong	A non-negative 64-bit integer number.

Table 4.4. Predefined XML Schema Datatypes that are Derived from Primitive Types

As with similar decisions in programming in general, the choice of which predefined data type to use as the basis of a schema-defined type is often arbitrary (e.g.,

the int or integer primitive XML types are largely interchangeable within AVCL schema). In the case of the exemplar data model of this research, decisions concerning which XML data types to utilize as the basis for schema-defined AVCL simple types were based on the heuristics of the data-binding software—the Sun Microsystems' JAXB API Specifically, XML numerical data types were chosen that JAXB binds to primitive types in the Java programming language (double and int in most cases).

An exemplar subset of the schema-defined simple datatypes from AVCL are summarized in Table 4.5. Numerical types defined by the schema are utilized in multiple places. Commands that explicitly set vehicle control actuators, for example, utilize either the percentType or signedPercentType regardless of the actuator's characteristics or potential vehicle-specific methods of specifying its setting (the value specifies the percentage of maximum actuator authority that is being commanded). The percentType is used for actuators with settings that are always positive (e.g., a UAV's engine power). The signedPercentType, on the other hand, is used to command actuators whose settings can have values that are either positive or negative (e.g., a UUV's rudder deflection).

Schema-defined string types in AVCL are far less general in their use. Normally appearing in only one or two places within the schema, strings provide a reader-friendly format for specific data items that might be equivalently implemented as integers. The reportingCriteriaType, for instance, is used to tell the vehicle when to transmit status reports as it attempts to accomplish the specified goals and has potential values of "never," "periodic," "statusChanged," "onCommence" or "onComplete." This particular type can be equivalently represented with the integers zero through four, but the syntactic sugar of the string type makes the resulting XML documents more readable by human operators.

A full description of all AVCL-defined simple datatypes can be found in Appendix A. It is worth emphasizing that a data model of the type implemented in this research implicitly relies on rigorously defined datatypes even though the details of individual type definitions themselves are often arbitrary in nature. The preceding discussion provides an overview of some of the considerations regarding the definition of

simple datatypes for an XML Schema-governed data model that is intended to be compatible with arbitrary application-level data models in a specific domain.

AVCL Simple Type	XML Base Type	Description
positiveIntType	int	A 32-bit integer with a value greater than zero.
nonNegativeIntType	int	A 32-bit integer with a value greater than or equal to zero.
clockHoursType	unsignedByte	Possible military time hour values (0 to 23).
clockMinutesType	unsignedByte	Possible wall clock minutes or seconds values (0 to 59).
timeZoneType	byte	Possible time zone offsets (hours) to Greenwich Mean Time (-12 to 12).
positiveScalarType	double	Double precision floating point number with a value greater than zero.
percentType	double	Double precision floating point number with a value between 0.0 and 100.0.
signedPercentType	double	Double precision floating point number with a value between -100.0 and 100.0.
latitudeType	double	Double precision floating point number representing a geographic latitude (-90.0 to 90.0).
longitudeType	double	Double precision floating point number representing a geographic longitude (-180.0 to 180.0).
headingType	double	Double precision floating point number representing a direction (0.0 to 360.0 degrees).
datumTypeType	string	Enumerated list of search datum types (e.g., point or area).
sensorTypeType	string	Enumerated list of sensor types.
trackModeType	string	Enumeration for potential navigation modes between waypoints.
reportingCriteriaType	string	Enumeration defining when status reports are required.
acknowledgeType	string	Enumeration defining whether a message needs acknowledgement.

Table 4.5. Exemplar AVCL Schema-Defined Simple Datatypes and the Predefined XML Datatypes from which they are Derived

3. Task-Level Behaviors

As discussed briefly in Chapter I, the set of task-level behaviors is a key component of the proposed common data model for autonomous vehicle operations since virtually every other aspect of this research relies on task-level behaviors. Task-level behavior definition and implementation is therefore a crucial piece of this research.

The assertions that vehicles of a particular type operate in more or less the same manner and that arbitrary vehicles can have their operations described using a single

vocabulary are not new. The Platform, Manipulator and Environment Sensor Subgroups of the JAUS Command Class message set (partially listed in Table 2.3), for instance, amount to a set of command messages that is potentially applicable to a broad array of vehicles (JAUS, 04-4). This command set is a direct (and probably inevitable) byproduct of the development JAUS as a vehicle-independent, open architecture designed around a message set and a fixed set of software components. An important aspect of the JAUS command set is that it is suitable for vehicles in multiple classes despite the fact that it was initially designed primarily for ground vehicles. That is, the commands are applicable not only to UGVs, but also to USVs, UAVs, and UUVs.

Of particular interest to this research are the commands of the JAUS Platform message subgroup (Table 4.6) because they deal with vehicle maneuvering. Each of these commands implicitly falls into one of four categories: open-loop, closed-loop / open-ended, closed-loop / terminating, or miscellaneous. Commands such as the Set Wrench Effort command, used to set the vehicle's six-degree-of-freedom propulsive effort, are open-loop commands. Commands of this sort explicitly set vehicle actuators without regard to the results—they simply tell the vehicle how fast to spin a propeller or how far to deflect a rudder. Further, there is no criteria by which to gauge the completion of the command, so open-loop commands remain in effect until superseded. Closed-loop / open-ended commands direct the maintenance of one or more vehicle-state characteristics and implementation requires state feedback to the controller. For instance, Set Global Vector commands the vehicle to maintain a specific forward speed, altitude, and orientation. A vehicle controller requires current speed, altitude and orientation information to make actuator adjustments to maintain the commanded state. As with open-loop commands, there are no predetermined completion criteria for closed-loop / open-ended commands, so once issued they remain in effect until superseded. Closed-loop / terminating commands are similar in that implementation requires state feedback, however commands of this type have implicit completion criteria. The most obvious JAUS command in this category is the Set Global Waypoint command that directs the vehicle to travel to a specified location. Once the commanded location has been reached, the command is complete and the vehicle reverts to some other form of control (the JAUS Reference Architecture does not specify a fallback control mode). Finally,

miscellaneous commands do not directly affect vehicle control, but set vehicle switches and parameters that may indirectly affect control. The only JAUS Platform subgroup message falling into this category is the Set Discrete Devices command which is used to command the status of the vehicle's propulsion system and associated subsystems. While not explicitly addressed in the JAUS Reference Architecture, the concepts of open-loop versus closed-loop and terminating versus open-ended commands are inherent aspects the command set. Not surprisingly, they are also relevant to other efforts in the area of vehicle-independent command.

Message Name	Command Category	Command Description
Set Wrench Effort	Open-Loop	Commands the propulsive effort and / or braking effort in each of 6 potential degrees of freedom.
Set Global Vector	Closed-Loop / Open-Ended	Commands the forward speed, altitude and orientation (bank, pitch and roll).
Set Travel Speed	Closed-Loop / Open-Ended	Commands the vehicle's forward speed.
Set Global Waypoint	Closed-Loop / Terminating	Commands the vehicle to proceed to a specific position (latitude / longitude, altitude and orientation).
Set Global Path Segment	Closed-Loop / Terminating	Commands the vehicle to travel a specified path.
Set Discrete Devices	Miscellaneous	Commands all controllable settings of the vehicle's propulsive system.

Table 4.6. JAUS Command Class Platform Subgroup Messages

The generic behaviors of CCL are somewhat more directly in line with the common autonomous vehicle data model task-level behavior set than the JAUS command set. In general, a behavior is a "definable unit of (normally low-level) activity initiated by certain inputs which generates certain outputs as a result of its activity" (Turner, et al., 93). A generic behavior is simply a behavior that is intended to be applicable across a spectrum of different vehicle types. Envisioned as the basis for a vehicle-independent UUV control architecture, eight potential functional categories of generic behaviors were initially explored (Turner, et al., 93):

- Low-level movement: attractive and repulsive behaviors
- Medium-level movement: obstacle avoidance, physical boundary following, wandering, etc.
- Sensor centric: assess sensor operation, modify sensing mode, etc.

- Effector centric: assess effector operation, modify effector operating mode, etc.
- Homeostatic: monitor operational envelopes, monitor energy budget, maintain trim, etc.
- Navigational level: path planning, situational assessment, searching, exploring, etc.
- Mission level: plan mission, make maps, assess threat, cooperate (with other agents), etc.
- Communication: message-level and discourse-level (i.e., conversation-level) communication

As a result of efforts to determine both a level of behavior abstraction appropriate for mission definition as well as the lowest level of behavior that can reasonably be considered hardware independent, the preceding list of functional categories for generic behaviors eventually evolved into the nine categories of Table 4.7 during the development of CCL (Komerska, et al., 99-2). CCL-related efforts to date have been focused on defining and implementing commands of the maneuver, navigate, communicate, configure, monitor, and execute convention categories. Commands of these categories are used to direct vehicle motion, monitor and react to internal and external events, and specify interactions between vehicles.

Functional Category	Description
Maneuver	Move or relocate in a certain manner.
Navigate	Update the vehicle internal reference location.
Communicate	Send (or request) information about the vehicle's state or understanding of the world.
Configure	Change a preconfigured aspect of the vehicle.
Monitor	Monitor an aspect of the vehicle and perform actions if specified criteria are met.
Execute Convention	Carry out a universally understood action.
Acquire Samples	Collect samples from the external environment.
Sense	Manipulate, configure, or utilize the vehicle's sensors.
Manipulate	Manipulate, configure, or utilize the vehicle's effectors.

Table 4.7. Generic Behavior Functional Categories of the CCL (After: Komerska, et al., 99-2)

In addition to the functional categories Table 4.7, the evolution of CCL provides an expanded analysis of behavior execution termination and the associated implementation issues. As previously noted different types of behaviors have inherently

different durations or time arcs over which they are active. In general there are three possible time-arc characterizations for a behavior. Some behaviors (such as those affecting device or vehicle settings) have very specific time arcs. Other behaviors have indeterminate but finite time arcs. Referred to as terminating in the previous discussion, behaviors of this sort have specific criteria that dictate success and the behavior's time arc ends when all criteria have been met. Finally, some behaviors can have potentially infinite time arcs. The open-loop and closed-loop / open-ended commands of the previous discussion fall into this category. From the standpoint of a vehicle-independent command language or data model, the various conditions under which behaviors can be terminated necessitates design decisions to avoid ambiguity and facilitate real-world use. In particular, it is probably inadvisable to allow behaviors with potentially infinite time arcs to remain active indefinitely. Additionally, it is possible for commands with indeterminate but finite time arcs to fail (e.g., a transit point might be unachievable, or the sea state might be too high to allow a GPS fix to be obtained). For this reason it is probably advisable to preclude even behaviors that normally terminate from remaining active indefinitely. Most vehicle command languages, including CCL and AVCL, prescribe the use of user-specified or default time outs for commands or behaviors that are not guaranteed to terminate. JAUS is an exception to this practice, probably a result of its extensive use to date in remotely operated vice autonomous vehicles.

Generic behaviors implemented in CCL along with their specified terminating criteria are listed in Table 4.8. One observation that can be made concerning CCL's set of generic behaviors is that there are no open-loop behaviors and only one closed-loop / open-ended behavior (Transit). This is a direct result of efforts to develop a hardware-independent behavior set that provides a level of abstraction appropriate for the mission level. In contrast the AVCL task-level behavior set does include a number of open-loop behaviors. In practice, however, these types of behaviors are rarely utilized by actual autonomous vehicles outside of systems testing (again, the use of open-loop commands in JAUS is a result of its use in remotely operated vehicles). The availability of open-loop commands in AVCL, therefore, is not meant to imply that they are a necessary component of a common data model. Nevertheless their commonplace use in human-in-the-loop systems makes them a sensible addition to the AVCL repertoire.

Generic Behavior	Functional Category	Description	Termination Criteria
GoTo	Maneuver	Transit to the specified location.	Destination reached or time out
MaintainPosition	Maneuver	Stay at the specified position.	Time out
Transit	Maneuver	Travel in the specified direction at the specified speed.	Time out
GPSFix	Navigation	Obtain a global positioning system fix.	GPS fix obtained or time out
AvoidRegion	Navigation	Do not enter the specified area.	Explicitly cleared
CommunicateStatus	Communicate	Transmit current status message.	Message sent or time out
CommunicateCapabilities	Communicate	Transmit vehicle capabilities summary message.	Message sent or time out
CommunicateFile	Communicate	Transmit a requested file.	Message sent or time out
CommunicateParameters	Communicate	Transmit vehicle parameters message.	Message sent or time out
CommunicateMessage	Communicate	Transmit a text message.	Message sent or time out
ConfigureParameters	Configure	Modify the specified vehicle parameters.	Parameter configured
MonitorParameter	Monitor	Monitor a specified parameter and report the values when directed.	As scheduled
SystemAdmin	Execute Convention	Alter the specified system administrative aspects.	Command executed
ModifyBehavior	Execute Convention	Modify the currently executing task or mission.	Command executed

Table 4.8. CCL Generic Behaviors, Functional Categories, and Termination Criteria (After: Komerska, 05)

A partial listing of AVCL task-level behaviors can be found in Tables 4.9 through 4.12. Task-level behaviors not included in this summary apply primarily to simulations and do not affect mission execution; these behaviors are covered in Appendix A. With the exception of the open-loop commands, most AVCL task-level behaviors are similar to generic behaviors of the CCL maneuver, navigate and communicate functional categories. The semantics of individual behaviors differ somewhat however. For instance AVCL provides a number of closed-loop / open-ended behaviors to control individual state values, whereas CCL provides only the Transit generic behavior. Conversely, CCL defines multiple behaviors in the communications functional category, while AVCL utilizes a single SendMessage behavior for all required communication.

Additionally, CCL does not implement behaviors that correlate to many of AVCL's miscellaneous task-level behaviors, most notably the Wait and WaitUntilTime behaviors.

Task-Level Behavior	Description	Next Behavior Issue Criteria	Termination Criteria
Waypoint	Transit to a location.	Behavior complete	Destination reached or time out
CompositeWaypoint	A parametrically specified pattern of waypoints.	Behavior complete	Last waypoint reached or time out
Hover	Proceed to a specified location and hover (UUV only).	Behavior complete	Destination reached or time out
Loiter	Proceed to a specified location and remain in the vicinity.	Behavior complete	Destination reached or time out
TakeStation	Maintain position at a specified range and bearing from an object.	Behavior complete	Destination reached or time out
Recover	Proceed to recovery station at the specified position.	Behavior complete	Recovery complete

Table 4.9. AVCL Closed-Loop / Terminating Task-Level Behaviors that have Implicit Termination Criteria

Task-Level Behavior	Description	Next Behavior Issue Criteria	Termination Criteria
MakeAltitudeAGL	Maintain the specified altitude above ground level (UAV only).	Immediate	New vertical command
MakeAltitudeMSL	Maintain the specified altitude above mean sea level (UAV only).	Immediate	New vertical command
MakeAltitude	Maintain the specified altitude above the sea floor (UUV only).	Immediate	New vertical command
MakeDepth	Maintain the specified depth below the sea surface (UUV only).	Immediate	New vertical command
MakeHeading	Maintain the specified heading.	Immediate	New heading command
MakeSpeed	Maintain the specified forward speed (m/sec).	Immediate	New speed command
MakeKnots	Maintain the specified forward speed (knots).	Immediate	New speed command
MoveLateral	Move laterally at the specified speed (cross-body-thruster UUV only).	Immediate	New Horizontal control mode
MoveRotate	Rotate about the vehicle's Z axis (cross-body-thruster UUV).	Immediate	New horizontal control mode

Table 4.10. AVCL Closed-Loop / Open-Ended Task-Level Behaviors Requiring State Feedback Control for an Indeterminate Period of Time

Task-Level Behavior	Description	Next Behavior Issue Criteria	Termination Criteria
SetAileron	Set the aileron deflection to the specified percent of maximum (UAV only).	Immediate	New Horizontal control mode
SetElevator	Set the elevator deflection to the specified percent of maximum (UAV only).	Immediate	New speed command
SetPlanes	Sets the deflection of the horizontal planes to the specified percent of maximum (UUV only).	Immediate	New vertical command
SetPower	Sets the forward propulsion power to the specified percent of maximum.	Immediate	New power command
SetRudder	Sets the rudder to the specified percent of maximum.	Immediate	New horizontal control mode
SetBodyThruster	Sets the cross-body thruster power to the specified percent of maximum (cross-body-thruster UUV only).	Immediate	New horizontal control mode

Table 4.11. AVCL Open-Loop Task-Level Behaviors that Remain Active for an Indeterminate Period of Time

Closely related to behavior time-arc termination is the issue of when to activate a behavior (i.e., at what point in the mission's execution do the specified behaviors become active). Unlike the commands of a typical sequential script or the tasks of a hierarchical controller, multiple behaviors will often be active simultaneously. The default behavior activation heuristic in CCL is to activate behaviors sequentially (i.e., not in parallel), as if the behavior sequence were a script. If multiple behaviors are processed at the same time, the first one is activated. Once the termination criteria of the behavior is met, the next behavior is activated. Alternatively, behaviors can be scheduled to activate when user-specified criteria are met. Starting criteria can be as simple as an absolute, relative or periodic time or can be based on the values of one or more monitored parameters. (Komerska, 05)

Task-Level Behavior	Description	Next Behavior Issue Criteria	Termination Criteria
Wait	Continue all current control modes for a specified period of time.	Behavior complete	Time out
WaitUntilTime	Continue all current control modes until a specified time.	Behavior complete	Time out
GpsFix	Obtain a global positioning system fix.	Behavior complete	GPS fix obtained or time out
SendMessage	Transmit a message to another vehicle or control station.	Immediate	Message sent
SetTime	Reset internal vehicle time to the specified time.	Immediate	Time reset
SetStandoff	Set the acceptable distance error for location capture.	Immediate	Standoff reset
SetPosition	Reset the internally maintained vehicle position to the specified location.	Immediate	Position reset
MissionScript	Replace the current task-level behavior script with one loaded from a specified file.	Behavior complete	Script loaded
MissionScriptInline	Load a new task-level behavior script from a specified file and include it in the current script.	Behavior complete	Script loaded
Quit	Shut down all vehicle systems.	Behavior complete	Shutdown complete

Table 4.12. Miscellaneous AVCL Task-Level Behaviors

AVCL takes a slightly different approach to the initiation of scripted task-level behaviors. As with CCL behaviors lacking scheduling criteria, AVCL task-level scripts are executed in order. However, for a some AVCL task-level behaviors, it is inappropriate to wait until a behavior terminates before activating the next behavior. In particular a number of individual AVCL task-level behaviors affect only a subset of the vehicle's controllable parameters and do not provide any guidance for how the vehicle is to control other aspects of its overall behavior (e.g., a behavior may direct a UUV to maintain a specific heading while leaving depth and speed unspecified). In these cases it makes sense to activate the next scripted behavior immediately.

AVCL bases its heuristic for scripted task-level behavior activation on the previously activated behavior. When reasonable, primarily in the case of closed-loop / terminating behaviors, a behavior is allowed to terminate before the next behavior is activated. In other cases, for instance when encountering open-loop or closed-loop /

open-ended behaviors, the next behavior is activated immediately (even if it supersedes and therefore terminates the previously activated behavior).

The default AVCL behavior activation heuristic can be effectively overridden with the Wait or WaitUntilTime behaviors, both of which inhibit further behavior activation until they terminate. When used in conjunction with closed-loop / open-ended behaviors, the Wait and WaitUntilTime have the effect of driving the vehicle along a trajectory for a specified period of time without regard to the vehicle's location upon behavior termination (similar to the JAUS Set Global Vector command or the CCL Transit generic behavior). When used in conjunction with closed-loop / terminating behaviors, the expected vehicle behavior is to maintain the location directed by the closed-loop behavior for the specified period of time.

The preceding discussion is not intended to constitute an exhaustive examination of the AVCL task-level behavior set (further discussion is left to Appendix A), but it does provide a comparison of the AVCL task-level behaviors and the CCL generic behaviors (and to a lesser extent the JAUS Platform message subgroup). Given the similarity of purpose, it is not surprising that AVCL behaviors and CCL behaviors share many characteristics. These similarities support the assertion that all activities undertaken by an autonomous vehicle of a given type come from a finite set of capabilities and that this set is essentially the same for all like vehicles. Further, it illustrates that these tasks can be identified, described and ultimately utilized to express tasking for arbitrary vehicles. The differences, however, illustrate that the suitable set of simple tasks is far from unique and that the data-model designer has some leeway in deciding the makeup of the set (i.e., the definition of the individual tasks comprising the set) as well as how and under what circumstances tasks are executed. For this reason the AVCL task-level behavior set is proposed as a potential vehicle-independent task set for expressing vehicle tasking, but no implication is made that it is the only or best task set available for this purpose. Within the larger context of the common data model it is important only that a suitable command set be developed and implemented. The AVCL task-level behavior set meets this requirement and effectively supports the other aspects of this research, in particular the equivalent mapping of AVCL behaviors to and from other command sets.

4. Declarative Task Specification

A common approach to realizing robust vehicle capability is from the bottom up—that is, combining low-level behaviors into increasingly complex aggregate behaviors. This is, the approach taken, for example, by CCL. Individual generic behaviors are defined and combined to form more complex behaviors that in turn can be used to define still-more complex behaviors. In the current development implementation of CCL, the kΩ planner running in the Distributed Control Environment software environment uses the current world model and the values of the monitored parameters to determine which behaviors need to be active. Additionally, the ability exists to instantiate new behaviors and modify existing ones as required to adapt to a changing environment (Duarte, et al., 05).

The data model developed in the course of this research supports complex autonomous vehicle operations through a completely different mechanism. Rather than requiring complex behaviors to be generated from the bottom up, AVCL provides for a description of the desired outcome, or goals, of an autonomous vehicle mission. This description, referred to as an agenda, is then used to plan behavior sequences to achieve the goals (derivation of behavior sequences is described in Chapters VI and VII). This method of describing a mission and the ability to convert it into a task-level behavior sequence can be applied in two ways. First, if used as part of a mission-planning system, a behavior sequence can be generated and translated into a vehicle-specific format for use in an actual vehicle. Although use of a declarative agenda in this manner does not allow a vehicle to adapt to a changing environment, the clarity of declarative missions offers advantages over manual generation of vehicle-specific scripts. The second way of using a goal-based mission description is as the input to the top-level of a multi-layer control architecture. When used in this manner, the data-model-compliant portions of the control architecture (i.e., the upper layers) use current knowledge about the world state to plan an appropriate task-level behavior sequence. This behavior sequence is ultimately converted into a vehicle-specific format for execution by the lower layers of the architecture (installation of a multi-layer AVCL-based controller on a non-AVCL vehicle is described in Chapter VII).

Goal-based declarative agenda descriptions in AVCL take the form of a binary finite state machine. Each state represents a single goal that the mission is to accomplish. State transitions occur when the vehicle successfully achieves a goal or fails to do so (e.g., exceeds allotted time or experiences a system failure that precludes goal success). The declarative mission description includes the start state (i.e., the first goal that the mission is to attempt), the vehicle's intended launch and recovery positions, and a list of areas that the vehicle is to avoid entering. The agenda is complete when a goal concludes (successfully or unsuccessfully) and there is no transition associated with that goal's success or failure. Figure 4.2 graphically depicts a UUV mission specified in this manner. In the example, the UUV will first attempt to search Area A. If successful, the vehicle will attempt to sample the environment of Area A (upon failure the vehicle will attempt to search Area B). If the area is successfully sampled, the vehicle will proceed to and search Area B. The mission will proceed from goal to goal in this manner until encountering a goal from which there is no transition specified. At this point the vehicle will transit to the recovery location and conduct any mission completion activities.

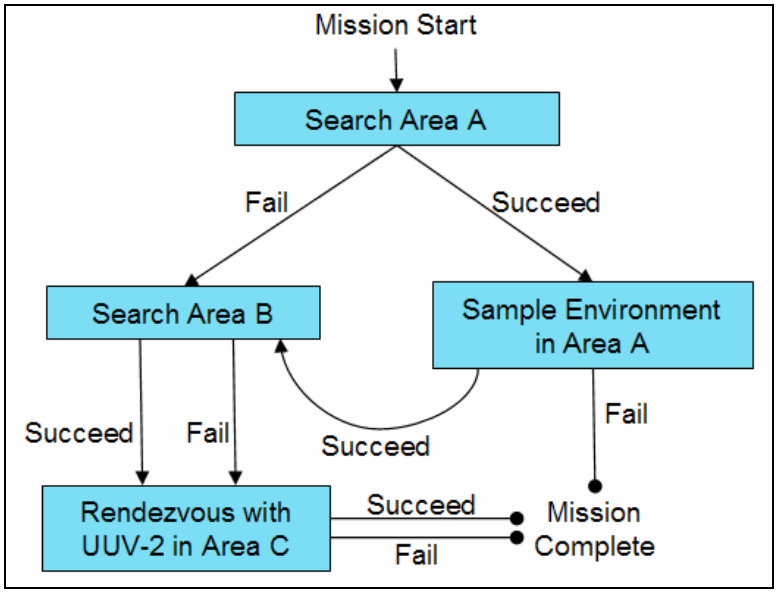


Figure 4.2. A Finite State Machine Representing the Goals and Mission Flow of an Exemplar Declarative UUV Agenda

In order to be of practical use, it is desirable for the data model to include not only a mechanism for describing mission flow, but a set of goal types for expressing specific goals. Similar to the task-level behavior set in that they are to be vehicle-independent,

the goal set consists of potential high-level mission objectives such as area search or rendezvous with another vehicle for data transfer. Some insight into the types of mission objectives that are potentially assignable to autonomous vehicles can be found in documents such as the Joint Robotics Program's Unmanned Ground Vehicle Master Plan (JRP, 04), the U.S. Navy's tactical memorandum regarding the integration of unmanned vehicles into maritime operations (CNO, 04) and the NAS study of the same subject (NAS, 05). Also applicable are JC3IEDM sections relating to action specification, since the command and control information that these portions of the model convey similar information as the goals of the data model. Because compatibility with external command and control systems is a design goal, it is reasonable to use a mechanism similar to JC3IEDM to specify goals in AVCL.

In this context, further examination of the JC3IEDM is warranted. JC3IEDM enumerates a total of 188 action types that can be represented by the model and utilized to provide tasking or summarize unit activities (MIP, 03-1). Most of these are not applicable to the autonomous vehicle domain and can be discarded. Ultimately, the 14 JC3IEDM action types of Table 4.13 were utilized as the basis for the AVCL goal set. Although other action-task activities are potentially applicable to autonomous vehicles, these activity types are proposed as a starting point for expressing tasks that autonomous vehicles might reasonably be expected to execute. This mapping does not imply, however, that all autonomous vehicles are capable of achieving goals of any type. For instance, it is unreasonable to expect an unarmed vehicle to successfully complete an Attack goal or a vehicle without electromagnetic sensors to intercept electromagnetic transmissions. Such cases are dealt with during the translation process to ensure that vehicles are not inappropriately tasked.

Each AVCL goal type listed in Table 4.13 has a number of common data parameters. For instance the instantiation of any goal includes the subsequent goals to execute upon successful or unsuccessful completion (i.e., the outbound transitions of the finite state machine state). Additionally, each goal must specify the operating area and timing constraints (start and end times or a maximum duration). Finally, each goal may include reporting criteria that dictate when status messages are to be transmitted (e.g., periodic, start and finish, any status change, etc.).

JC3IEDM Action Task Activity	Corresponding AVCL Goal Type	Description
Attack, not otherwise specified	Attack	To conduct a type of offensive action characterized by employment of firepower and maneuver to close with and destroy an enemy.
Decontamination Services	Decontaminate	To provide purification making an area safe by absorbing, destroying, neutralizing, making harmless, or removing chemical, biological, or nuclear contamination.
Demolish	Demolish	To destroy structures, facilities, or material by any available means.
Illuminate	IlluminateArea	To provide battlespace lighting by searchlight or pyrotechnics.
Intercept	MonitorTransmissions	To conduct electronic warfare support operations with a view to searching, locating, recording and analyzing radiated electromagnetic energy.
Jam	Jam	To deliberately radiate, re-radiate or reflect electromagnetic energy with the object of impairing the use of electronic devices or systems.
Mark	MarkTarget	To make visible (by the use of light, infrared, laser, smoke, etc.) of an object in order to allow its identification by another object.
Move	Reposition	To change position from one location to another.
Patrol	Patrol	To gather information or carry out a security mission.
Rendezvous	Rendezvous	Achieve a meeting at a specified time and place.
Sample, biological	SampleEnvironment	Collect environmental samples for testing for biological hazards.
Sample, chemical	SampleEnvironment	Collect environmental samples for testing for chemical hazards.
Sample, nuclear	SampleEnvironment	Collect environmental samples for testing for nuclear hazards.
Search	Search	To look for lost or unlocated objects or persons.

Table 4.13. JC3IEDM Action-Task Activities Incorporated into AVCL as Declarative Agenda Goal Types

In addition to their common characteristics, individual goal types may have specific parameters unique to that particular type of goal. A Search goal, for instance, must specify the desired probability of detection, whether the datum (the most probable location of the object of the search) is a point or an area (i.e., is the search to focus on the centroid of the operating area or to apply equal attention across the entire area), and whether the objective of the search is a single or multiple entities. Additionally, an instantiated Search goal can specify one or more search targets. A Reposition goal, on

the other hand, requires no information beyond that common to all goal types. A full description of the parameters associated with each goal type can be found in Appendix A. Parameters for each goal type are intended to provide the information necessary to adequately interpret the goal and to support automated plan generation.

5. Mission Results

A peripherally related portion of the data model that was developed in parallel with task-level and declarative mission specification deals with accumulated mission results. Not surprisingly, different vehicles use different data formats to encode mission data such as vehicle telemetry, control settings, and event data. It can be argued that since mission results are not directly relevant to tasking or communications they do not need to be included in the proposed common data model. While this is true to an extent, there are also factors that argue for its inclusion. Most importantly, messages containing telemetry and event data are among the most common in inter-vehicle communications so these types of data must be included in the model's communications mechanics. Thus, it is reasonable to define a mission-results data-model subsection and reuse its components for message payload (the same approach is utilized for communication of task-level behaviors or declarative goals). On a more subjective level, a vehicle-independent format for mission results can facilitate the comparison of results from dissimilar vehicles.

All mission-results data in AVCL falls into one of two categories: discrete samples of continuous vehicle state information or asynchronous event information. The first category includes vehicle telemetry (position, velocity, etc.) and control settings. Data of this sort amounts to a time-stamped snapshot of a continuous data stream corresponding to a specific point in time. The second category consists of events that can occur at any time during a mission. Generated contacts, messages sent and received, and systems failures all fall into this category. Discrete samples of continuous data are maintained in a generalized mission results section of an AVCL document while discrete event data is maintained in an event log in the same document. Appendix A provides a full description of all potential mission results and mission log content.

6. Inter-Vehicle Communications

Inter-vehicle communications is a broad area with significant ongoing research. Topics range from networking and data-transmission aspects of autonomous vehicle

communications (e.g., ad hoc networks and forward error correction) to more abstract topics such as behavior and protocol requirements for cooperation. Of particular interest from the common data model standpoint are the types of messages required to support cooperative autonomous vehicle behaviors, since these are the messages that must be accounted for in the data model.

Existing research, encapsulated effectively in the FIPA Communicative Act Library Specification, indicates that communicative acts (which take the expressive form of messages when applied to autonomous vehicles) between autonomous agents can be classified as one of two types: request or inform. In fact, all message types within the Communicative Act Library Specification are derived from either the request or inform base types (FIPA, 02). Request messages are those that request either information (e.g., vehicle state or contact information) or action (e.g., an individual command or an entire mission) from the receiving vehicle. Inform messages, on the other hand, provide information such as the sending vehicle's state or sensor data. The Communicative Act Library Specification rigorously defines the semantics of its 22 acts, but does not specifically define the content. Rather, it provides a wrapper for the content that describes how it is to be interpreted. With the exception of CCL which expresses all messages as either command or inform (Komerska, 05), existing autonomous vehicle communications schemes do not directly cite the FIPA Communicative Act Library Specification, but each of them implicitly utilizes this request / inform model.

Designed to support a broad array of autonomous agents in addition to autonomous vehicles, the 22 messages of the current Communicative Act Library Specification provide more functionality than is required to fully exercise the capabilities of available and developmental vehicles. Current autonomous vehicle communication schemes, therefore, are not required to implement full library functionality. In addition the syntax and format is not generally utilized in autonomous vehicle messaging. Instead, autonomous vehicle communications schemes tend to implement only those messages that meet the specific requirements of the envisioned vehicle interactions.

Concerning the more general messaging requirements of inter-vehicle communications, a number of previously cited efforts are relevant. For instance JAUS

and C2L are both designed around the content of their respective message sets. Worth noting in the case of JAUS are the messages of the Query and Inform message classes as well as the previously discussed Platform class. Each of these message classes contain Platform, Manipulator and Environment Sensor subgroups. Further, each message of the Platform class subgroups can be correlated with one or more messages in the same subgroup of the other classes. That is, for every Platform class message, there are one or more Query Class messages that can be used to request information about the commanded parameters and one or more Inform Class messages that are used to provide the appropriate information (JAUS, 04-4). In addition the Query and Inform classes contain messages used to request and provide vehicle specifications and capabilities. The JAUS message set fairly clearly illustrates the request-inform messaging model—Platform and Query classes form JAUS' request messages while the Inform Class defines its inform messages. (JAUS, 04-4)

The C2L specification defines messages for conveying much of the same information as JAUS—directive commands and vehicle status. Messages for requesting information are conspicuously absent from the C2L message set. Messages are available for transmitting vehicle state for a number of different vehicle types and also for transmitting various types of sensor data, but no message types are included to request this information. Thus, C2L has a somewhat narrower implementation of the request-inform communications model than JAUS. Since the conditions upon which inform messages are transmitted are defined outside of the scope of C2L, they can vary from vehicle to vehicle or operation to operation.

These two examples, as well as CCL's use of messaging to command generic behaviors and propagate data, make it evident that among the most basic requirements of inter-vehicle communications is the ability to exchange command and state information. For a multi-vehicle system in which the composition is predefined (e.g., a JAUS system) or in which the capabilities of all vehicles are known (e.g., a system of C2L-compliant vehicles) the predefined message types are probably sufficient. On the other hand, they will likely prove insufficient for systems consisting of arbitrary vehicles with unknown capabilities, or for systems whose composition may change over time. For these systems,

messages will be required to facilitate the discovery of vehicle capabilities as well as the formation and maintenance of vehicle groups.

Among the efforts addressing these aspects of multi-vehicle operations are the Meta-Level Organization and Task-Level Organization protocols developed through the Cooperative Distributed Autonomous Oceanographic Sampling Networks (CoDA) research (Chappell, et al., 97)(Turner and Turner, 04). CoDA applies a multi-agent systems and distributed artificial intelligence approach to the domain of autonomous vehicle operations. In the proposed CoDA system, vehicles self-organize into a Meta-Level Organization in order to discover resources and design a Task-Level Organization to fit the current situation and system composition (Turner and Turner, 04). Individual autonomous vehicles go through protocols to discover, join or leave an existing Meta-Level Organization, form a Meta-Level Organization if one does not exist, and form a Task-Level Organization to accomplish specific tasks. Inter-vehicle communications are used to facilitate Meta-Level Organization and Task-Level Organization formation and to maintain synchronization among vehicles as they proceed through their protocols.

Although the design and implementation of specific CoDA protocols are the subjects of ongoing research, a significant portion of the required messaging capabilities are becoming clear. Not surprisingly, the communications requirements of a system along the lines of CoDA include tasking and state messages similar to those available in JAUS and C2L. Also required are a set of messages specific to the tasks of organization formation and maintenance. Specifically, individual vehicles need to broadcast messages to locate an existing organization, initiate and conclude organization formation, dissolve an organization, or join or leave an existing organization. Finally, the formation and maintenance of autonomous vehicle organizations to support coordination requires messages that convey the capabilities of individual vehicles. The originally proposed CoDA protocols utilized has-capabilities and controls-capabilities messages for this purpose. More recently much of the CoDA functionality has been incorporated into CCL which conveys vehicle capability with a 36 byte data object that encodes 69 operating characteristics that are used to specify operational limits, vehicle physical characteristics and installed sensor and navigation systems (Komerska, 05).

Based on this discussion it is possible to itemize the minimum messaging requirements for a cooperative autonomous vehicle system (Table 4.14). The requirements can be broken down into five fairly broad categories, each of which can be categorized as a request or inform message per the FIPA Communicative Act Library Specification. All of the discussed communications languages define messages that can be used to command a vehicle to perform an action or to provide state information about a vehicle. Further, with the exception of C2L, each language provides messages that can be used by a vehicle to describe its own capabilities or to request state information from another vehicle. Finally, the languages and protocols that are designed to support dynamic cooperative groups require messages to support group maintenance.

Message Type Description	Message Category
Command the vehicle to perform an action	Request (action)
Request information from the vehicle	Request (information)
Provide vehicle state or event information	Inform
Provide vehicle capabilities information	Inform
Initiate or request cooperative group maintenance action	Request (action)

Table 4.14. Proposed Minimum Messaging Requirements to Support Multi-Vehicle Operations

A significant portion of the messaging capabilities listed in Table 4.14 are defined in other portions of the common autonomous vehicle data model defined by AVCL. The constructs required to command vehicle actions are available in the task-level behavior and declarative goal portions of the AVCL data-model and constructs required to support vehicle state and event information are available in the mission results portion. It is reasonable to reuse these constructs in command and inform message definitions. Thus, definition of specific message types is required only for information requests, group maintenance requirements and vehicle capability reporting. Information requests and requests for group maintenance are incorporated into the model as enumerated strings that identify the specific type of information or action that is being requested. Vehicle capability reporting is implemented using vehicle-type-specific elements that can be used to convey maneuvering limits, physical characteristics and sensor configuration. A more complete summary of the message types available in AVCL is provided in Table 4.15.

Message Type	Message Content	Implied Message Intent
Mission Specification	A complete mission (task-level behavior sequence or a set of declarative goals).	The sender is requesting that the recipient execute the mission.
Behavior Specification	A single task-level behavior.	The sender is requesting that the recipient invoke the behavior.
Group Maintenance	A group initiation or maintenance request.	The sender is attempting to locate, form, join or leave a group.
Information Request	A request for vehicle information.	The sender is requesting information from the receiver (e.g., vehicle state, sensor data or event summary, etc.).
Vehicle Characteristics	A capabilities summary.	The sender is reporting its physical and operational characteristics.
Vehicle State	Telemetry or control information.	The sender is reporting one or more aspects of its current state (location, speed, control settings, etc.).
Sensor Data	Sensor information.	The sender is providing its most recent sensor data.
Vehicle Event	An observable event.	The sender is reporting an event (e.g., contact report, engineering casualty, etc.).

Table 4.15. Message Types Incorporated into the AVCL Schema

C. SUMMARY

A brief analysis of the differences between ontologies and data models leads to the conclusion that the data model proposed and implemented in this research does not inherently possess a sufficient degree of semantic richness to be considered an ontology. Specifically, it constrains the format of the model, but does not explicitly define the nature of the relationships between the various model elements. Nevertheless it constitutes a rigorously defined data model that is sufficient to serve as a vehicle-independent vocabulary for autonomous vehicle operations.

The data-model developed in the course of this research defines a set of task-level behaviors that are suitable for use with arbitrary vehicles. They are also suitable as building blocks for use in completing more abstract declarative goals. Appropriate task-level behavior sequences can be automatically generated to accomplish the goals specified in a data-model-compliant declarative mission by either an off-line mission planning system or an on-vehicle high-level controller. In either case, the vehicle-independence of the task-level behavior sequence makes it suitable for conversion to arbitrary vehicle-specific formats. In addition to both scripted and declarative mission

specification, AVCL implements inter-vehicle communications and mission results. Subsequent chapters demonstrate the utility of all AVCL sections in support of arbitrary vehicle operations.

THIS PAGE INTENTIONALLY LEFT BLANK

V. VEHICLE-SPECIFIC LANGUAGE CONVERSIONS

A. INTRODUCTION

The discussion to this point has focused on the structure of the exemplar common autonomous vehicle data model. More important than the data model's design, however, is its application to actual vehicles. In a sense the data model amounts to infrastructure that enables the techniques explored in this research to support vehicles of various types. This chapter discusses one of the more important capabilities offered by an XML-based, common autonomous vehicle data model—automated translation between vehicle-specific data formats.

Specifically covered in this chapter is the conversion of five exemplar vehicle-specific data formats to and from AVCL. Text-based data formats include mission programming languages for the NPS Phoenix UUV, the NPS ARIES UUV, the Naval Oceanographic Office Seahorse UUV, and the Hydroid REMUS UUV. Additionally, the compatibility of AVCL with binary vehicle-specific formats is demonstrated through translations to and from the platform subgroups of the JAUS Command, Query and Inform message classes.

The JAUS message sets of interest have been discussed in sufficient detail previously and do not require further elaboration. The four text-based data formats, however, have not been introduced thus far. Therefore, the next section provides a brief overview of the structure and semantics of each of these data formats. The remainder of the chapter details the specific mechanisms by which AVCL documents are converted to and generated from each of the five exemplar formats.

B. RELATED MISSION PROGRAMMING LANGUAGES

1. The Phoenix UUV Command and Control

The Phoenix UUV was a research vehicle designed and built by the NPS Center for Autonomous Underwater Vehicle Research. The Phoenix possessed lateral and vertical cross-body thrusters that enabled it to hover in a fixed position, making it unique among the UUVs discussed here. Further differentiating Phoenix from other vehicles to which AVCL has been applied is the use of the RBM control architecture with high-level

missions defined in Prolog (Healey, et al., 96). AVCL compatibility with RBM missions defined in Prolog is not a specific goal of this research, although the increasingly common use of XSLT to generate program code and the template-based nature of the RBM mission planning expert system described in (Davis, 96) provide evidence that they are, in fact, compatible (in addition, a proposed extension to the RBM wherein an AVCL goal-based declarative mission definition replaces the Prolog definition is the subject of Chapter VII). The discussion in this chapter focuses on the application of AVCL to the behavior definition language that formed the underpinning of the original RBM (Brutzman, 94) (Davis, 96).

Although the Phoenix UUV is no longer operational, its command language was chosen as an AVCL conversion target for a number of reasons. First, the mapping between this language and AVCL is fairly straightforward making it a good candidate for the first target language. Additionally, the Phoenix tasking language is the only behavior-based language for which AVCL translations have been developed. Finally, since the Phoenix command language was designed within the context of a multi-layer control architecture, successful conversion between AVCL and the Phoenix command language serves to demonstrate the applicability of a common data model to vehicles utilizing various control architectures.

At the execution level, a Phoenix mission specification takes the form of a behavior script. Individual behaviors are defined by a keyword and a parameter list (a summary of the most common behaviors is provided Table 5.1). The keyword identifies the type of behavior being initiated. The number of parameters associated with the behavior determines how the individual parameters are interpreted. For example the waypoint behavior has four potential parameter arrangements: Cartesian coordinates, depth, propeller revolutions per minute during transit, and standoff distance; Cartesian coordinates, depth and standoff distance; Cartesian coordinates and depth; or just Cartesian coordinates. Behavior initiation and termination is handled in a manner similar to that used to control the activation and deactivation of AVCL task-level behaviors—activation is based on the preceding behavior type while termination is based on the activated behavior type.

Behavior Name	Description	Parameters
Depth	Set vehicle's commanded depth.	Commanded depth
GPS	Obtain a GPS fix.	None
Heading	Set vehicle's commanded heading.	Commanded heading
Hover	Command the vehicle to proceed to a specified geographic location and hover there.	Cartesian X coordinate of the destination Cartesian Y coordinate of the destination Commanded depth at hover point (optional) Heading to maintain during hover (optional) Standoff distance from destination (optional)
Lateral	Activate lateral body thrusters to slide sideways.	Commanded lateral thruster voltage
Mission-Script	Replace the currently executing mission script with a new one.	Path to the new mission script file
Planes	Open loop horizontal planes deflection command.	Commanded plane deflection
Position	Set the vehicle's internally maintained position.	Cartesian X coordinate Cartesian Y coordinate Depth (optional)
Quit	End the mission and shut down.	None
Rotate	Activate lateral body thrusters to rotate about the vehicle's Z axis.	Commanded lateral thruster voltage
RPM	Set ordered propeller revolutions per minute.	Both or left propeller revolutions per minute Right propeller revolutions per minute (optional)
Rudder	Open loop rudder deflection command.	Commanded rudder deflection
Thrusters-Off	Disable cross-body thrusters.	None
Thrusters-On	Enable cross-body thrusters.	None
Wait	Continue current behavior for a specified period of time.	Time to wait before commencing the next behavior
Wait-Until	Continue current behavior until the specified time.	Clock time at which to commence the next behavior
Waypoint	Command the vehicle to proceed to a specified geographic location.	Cartesian X coordinate of the destination Cartesian Y coordinate of the destination Commanded depth at destination (optional) Revolutions per minute to use during transit (optional) Standoff distance from destination (optional)

Table 5.1. Selected Phoenix UUV Behaviors (After: Davis, 96)

2. ARIES UUV Mission Specification

The follow-on vehicle to the NPS Phoenix is the NPS ARIES UUV and this vehicle is currently the primary research platform of the NPS Center for AUV Research. Although similar in many regards to the Phoenix, ARIES has numerous navigation, sensor, electronic and computational system upgrades (Nicholson, 04). Additionally,

ARIES no longer has cross-body thrusters installed so it is not currently capable of hovering at a fixed position. Most relevant from the standpoint of this research is that ARIES no longer utilizes the RBM control architecture as described in (Byrnes, 93). Rather, for most operations ARIES missions take the form of a simple waypoint script (Marco, 01).

From a conceptual standpoint, an ARIES waypoint list is the simplest mission specification format to which AVCL has been applied. The first line in the mission file consists of a single integer value that identifies the number of waypoints the vehicle is to execute. Each subsequent line contains 11 white-space-delimited numerical fields that describe a single waypoint. Individual fields specify the location of the waypoint, describe the vehicle's control modes en route, and define how much time the vehicle is allotted to successfully reach the waypoint (the content description of each field is provided in Table 5.2). The vehicle transits to waypoints in the order in which they are specified and does not begin transiting to a new waypoint until the preceding waypoint has been reached. Failure to arrive at any waypoint within the allotted time is a mission abort criteria and causes the vehicle to terminate the mission and surface.

Field	Description
1	Cartesian X coordinate of the waypoint (meters)
2	Cartesian Y coordinate of the waypoint (meters)
3	Left screw commanded speed (volts)
4	Right screw commanded speed (volts)
5	Vertical control mode flag: 0=Depth Control, 1=Altitude Control
6	Commanded altitude during transit (meters)
7	Commanded depth during transit (meters)
8	Perform a GPS popup during transit: 0=No, 1=Yes
9	Duration of GPS popup if commanded (seconds)
10	Watch radius (i.e., how close to the waypoint is good enough) (meters)
11	Maximum time allotted to reach the waypoint (seconds)

Table 5.2. Field Descriptions for Individual Entries of an ARIES Waypoint List (From: Marco, 01)

Given that a goal of the common data model is to capture the semantics of common autonomous vehicle operations in the task-level-behavior set, it is not surprising that the representation of ARIES waypoint scripts with AVCL is not difficult—transit between waypoints is, after all, one of the most frequently required autonomous vehicle

tasks. The self-contained nature of each waypoint (a byproduct of their specification as tasks vice behaviors) introduces a number of issues during the translation process (particularly when converting AVCL task-level behavior scripts to ARIES waypoint lists. Various techniques have evolved over the course of this research to facilitate the process which are discussed in detail later in this chapter.

3. Seahorse UUV Task Set

The Seahorse UUV is a long-endurance, oceanographic survey vehicle designed and built by the Pennsylvania State University Advanced Research Laboratory and operated by the Naval Oceanographic Office (Peterson and Head, 02). The Seahorse possesses control, navigation and sensor capabilities similar to those of ARIES, but uses its own command language to define mission scripts. Consisting of the six task types listed in Table 5.3, the Seahorse command language arguably provides more control than ARIES waypoint descriptions. It is, however, still a task-scripting language in which a new task is not initiated until the preceding task has completed.

Task Type	Description
Launch	Causes the vehicle to submerge and start the propulsion motor. Always the first order of a valid mission.
Waypoint Navigation	Directs the vehicle to transit from the current location to a specified new location.
GPS Fix	Directs the vehicle to surface, acquire a GPS fix, and return to the previously ordered depth or altitude above the bottom.
Station Keep	Directs the vehicle to maintain a circular holding pattern about a specified destination for a specified period of time.
Surface Comms	Directs the vehicle to surface at the current location for communications purposes.
Rendezvous	Directs the vehicle to proceed to a specified recovery position. Always the last order of a valid mission.

Table 5.3. Seahorse UUV Task Set (After: NAVO, 04)

In addition to providing more task types than ARIES, the Seahorse command language is more complex from a syntactic standpoint. Each task has a number of parameters and options that dictate how the task is to be accomplished. Further, positions, distances and speeds can each be specified in a number of ways. Keywords are used to indicate the parameters and units as shown in the example station-keeping order of Figure 5.1. The example demonstrates the use of parameter-type keywords (to the left of the ':' on each line) to specify the purpose of a given parameter (e.g., the latitude at

which to maintain station specified in the third line) with the parameter value following the parameter-type keyword. Additional keywords are used to specify the parameter units if applicable. The transit altitude and loiter depth of the example, for instance, are specified in meters, but the language permits their specification in feet as well. Other tasks use the same parameter-type / parameter-value pattern and in many cases one parameter type can be substituted for another (e.g., the transit altitude of Figure 5.1 can alternatively be specified as a transit depth).

Valid Seahorse mission files always begin with a launch order and finish with a rendezvous order, so for all practical purposes, there are only four order types available for general use—navigate to a waypoint, travel to and maintain a station, obtain a GPS fix and surface for communications. All of these tasks are easily expressed with AVCL task-level behaviors. As is the case with ARIES waypoints, however, each task is self contained which necessitates the use of similar programming patterns to convert Seahorse command files to and from AVCL task-level behavior scripts.

```
Start_Order               :   Station_Keep_Order
Scheduling_Info_Is_Timed  :   False
Destination_Latitude      :   28.1 Degrees
Destination_Longitude     :   88.1 Degrees
Until_when                :   90.0 Minutes
Transit_Altitude          :   10.0 Meters
Loiter_Depth              :   15.0 Meters
```

Figure 5.1. An Example Seahorse UUV Station Keeping Order (After: NAVO, 04)

4. The REMUS UUV Objective Set

Of the vehicle-specific data formats to which AVCL has been applied, the command language of the REMUS family of UUVs is the most structurally complex. The complexity arises from the use of references within mission objectives, multiple methods for specifying locations, and the requirement to designate the locations of at least two navigation transponders.

A REMUS mission file consists of a series of locations followed by a series of objectives. Locations are used to specify the geographic positions of navigation

transponders and also to define locations that can be referenced later in the file. Each position definition contains the "[Location]" keyword, a type, a label, and the geographic position. Locations defining transponder locations are assigned a type corresponding to the transponder. All other locations are assigned a type of "waypoint." The actual position can be specified using latitude and longitude or a reference to a previously defined position and can include an offset as well (i.e., a location can be defined relative to an existing location). The end of the location section of the mission file is designated by an empty location (i.e., a location with a waypoint type but no position).

Each objective in the mission file begins with the "[Objective]" keyword and its type (available objective types are listed in Table 5.4). REMUS objective definitions are similar to Seahorse tasks in that each objective description uses parameter-keyword / parameter-value pairs to describe the objective and how its execution is to proceed. Objectives with geographic parameters can reference locations from the locations portion of the mission file or define their own locations (and may utilize an offset from another position as well).

Objective	Type of Command	Description
Set Position	Initialization	Set the vehicle starting latitude and longitude (must be the first objective of the mission).
Wait Depth	Mission Start	Waits until the vehicle is deeper than a depth before commencing the mission.
Wait Prop	Mission Start	Waits for the vehicle's prop to be spun before commencing the mission.
Wait Run	Mission Start	Waits until a run command is received before commencing the mission.
Wait Magnet	Mission Start	Waits until the magnet is removed from a vehicle sensor before commencing the mission.
Navigate	Waypoint	Navigates to a waypoint using the best method.
Dead Reckon	Waypoint	Navigates to a waypoint by dead reckoning.
Transponder Home	Waypoint	Uses a transponder to home the vehicle to a specified position.
Navigate Rows	Waypoint (multiple)	Directs a lawn-mowing pattern of waypoints.
Surface	Miscellaneous	Surfaces the vehicle at the end of a mission.
Compass Cal	Miscellaneous	Performs an in-water compass calibration.
Include	Miscellaneous	Includes an external mission file into the currently executing mission.
End	Miscellaneous	Uses to indicate the end of a mission.

Table 5.4. REMUS UUV Objective Types (After: Hydroid, 02)

Each mission begins with a set position objective (and in most cases one of four mission start objectives) and concludes with an end objective (probably immediately preceded by a surface objective). This leaves six objective types that are used to define the bulk of most missions. Three of these are used to define a single waypoint (with the navigate objective being the preferred type since it allows the vehicle to dynamically choose the best navigation method). A fourth provides a way of defining a complete waypoint pattern with a single objective.

Despite the syntactic complexity, REMUS missions are executed as task scripts and are semantically similar to the mission files of the ARIES and Seahorse vehicles. In this regard the REMUS objective set is largely compatible with AVCL's task-level behaviors and the same programming patterns as with ARIES and Seahorse missions are applicable. On the other hand, the language contains a number of tasks that are unique to the REMUS family of vehicles that require special handling in the conversion process.

C. TEXT-BASED VEHICLE-SPECIFIC DATA FORMATS

1. Generation of Vehicle-Specific Documents from Data-Model-Compliant XML

a. Introduction

As discussed in Chapter III, XSLT is the mechanism of choice for converting XML data to other text-based formats so its use in converting AVCL to vehicle-specific formats is not surprising. The programming pattern used in XSLT stylesheets that process AVCL documents involves the implementation of a template for each AVCL element that correlates to the target data format—most frequently in the case of AVCL, the task-level behaviors. Depending on the characteristics of the specific behavior and those of the target data format, template actions might include the generation of vehicle-specific commands or simply the update of command parameters for use during subsequent behavior processing. For instance, all of the vehicle-specific languages discussed here define some form of waypoint command that can be used to direct the vehicle to specific geographic positions. Regardless of the target language, it is appropriate for the stylesheet's Waypoint-behavior template to generate a vehicle-specific waypoint command. On the other hand, only the Phoenix command language includes unique behaviors ordering depth below the surface or altitude above the bottom.

Rather, command parameters of this type are typically embedded within the vehicle-specific waypoint command. Stylesheets targeted to most vehicle-specific formats, therefore, utilize templates for the MakeDepth or MakeAltitude behaviors that update commanded depth or altitude for future use but do not generate output.

b. Conversion of AVCL for the Phoenix UUV

The first vehicle-specific language for which an XSLT stylesheet was implemented is the behavior-scripting language of the Phoenix UUV. Of the languages for which mappings to and from AVCL have been developed, Phoenix behavior scripts are the most semantically similar to AVCL task-level behavior scripts. Phoenix behavior activation and termination is handled in a manner similar to AVCL task-level behaviors and individual Phoenix behaviors are often similar enough in function to AVCL behaviors to enable a one-to-one mapping between the two languages. Of the 19 AVCL task-level behaviors listed in Table 5.5, 15 can be mapped to a single Phoenix behavior (column two of the table). The other four are mapped to a primary Phoenix behavior but may require the use of one or more additional Phoenix behaviors (listed in the third column of Table 5.5) to fully express their content.

AVCL Task-Level Behavior	Associated Phoenix Behavior	Possible Additional Required Phoenix Behaviors
CompositeWaypoint	Waypoint (multiple)	Depth, RPM, GPS, Standoff
GpsFix	GPS	None
Hover	Hover	Depth, Heading, GPS, Standoff
Loiter	Waypoint	Depth, RPM
MakeDepth	Depth	None
MakeHeading	Heading	None
MakeSpeed	RPM	None
MakeKnots	RPM	None
MissionScript	Mission-Script	None
MoveLateral	Lateral	None
MoveRotate	Rotate	None
Quit	Quit	None
SetPlanes	Planes	None
SetPosition	Position	None
SetPower	RPM	None
SetRudder	Rudder	None
Wait	Wait	None
WaitUntilTime	Wait-Until	None
Waypoint	Waypoint	Depth, RPM, GPS, Standoff

Table 5.5. A Partial Mapping of AVCL Task-Level UUV Behaviors to Phoenix UUV Behaviors

The similarity between AVCL task-level behavior scripts and Phoenix behavior scripts greatly simplifies the AVCL-to-Phoenix translation stylesheet. Most significantly, since individual behaviors do not necessarily override all previously ordered control parameters (e.g., a new depth can be ordered without affecting the ordered speed or heading), there is no requirement to maintain control parameter information outside the scope of an individual template. Stated differently, templates are completely self contained—their output is not affected by previously invoked templates and they do not influence subsequently invoked ones.

Translation of AVCL task-level behaviors amounts to the generation of one or more Phoenix behaviors that command the actions specified by the AVCL behaviors. AVCL behaviors such as MakeDepth are translated into a single Phoenix behavior (column two of Table 5.5). AVCL behaviors that potentially affect multiple control parameters are often handled differently. The AVCL Waypoint behavior, for example, can be used to specify not only a destination position, but the depth of the waypoint, the speed of transit and how close the vehicle must get to the destination (an en route GPS fix can be ordered as well). While the Phoenix Waypoint behavior is capable of commanding speed, depth and standoff distance as well, it cannot do so in arbitrary combinations (e.g., a transit speed can be commanded only if a depth is commanded as well). Given the various control parameter combinations that can be ordered by a single AVCL Waypoint behavior, it is more appropriate to use the Phoenix Waypoint behavior to specify only the horizontal location of the destination. Other control parameters specified in the AVCL behavior are converted to Phoenix behaviors that immediately precede the Waypoint behavior. Since the Phoenix behavior activation scheme calls for immediate activation of the behavior immediately following a Depth, RPM or GPS-Fix behavior, this behavior sequence is effectively equivalent to a single behavior encapsulating all parameters. The AVCL Waypoint behavior in Figure 5.2, for instance, is translated to a sequence of Phoenix behaviors specifying the transit depth and transit speed prior to the waypoint command.

Also illustrated in Figure 5.2 are the implementation of unit conversions within the XSLT stylesheet. The units used to specify the Cartesian coordinates and depth of the AVCL waypoint (meters) are converted to feet as required by Phoenix and

the propeller power of 75 percent is converted to an actual revolutions per minute value. While these particular conversions are easily accomplished with XSLT, they do highlight a potential unit conversion problem. AVCL allows positions to be specified using either Cartesian coordinates in an earth-fixed coordinate frame (with the origin located at a geographic position specified using a GeoOrigin element), a relative position (Cartesian coordinates in an earth-fixed coordinate system located at the current vehicle position) or latitude and longitude. Phoenix, on the other hand, accepts only Cartesian coordinates. Two methods are commonly utilized by applications requiring this type of geographic data processing. If a high degree of accuracy is required or applications work with actual cartographic products, Geographical Information System support is often required. On the other hand, many applications (including many autonomous vehicles and vehicle planning systems) directly implement formulas that provide the requisite level of accuracy without the overhead or cost of a Geographical Information System. Unfortunately, XSLT processors do not have inherent access to an installed Geographical Information System and XPath does not have built-in trigonometric functions required to implement conversions within the stylesheet.

AVCL Task-Level Behavior:

```
<Waypoint>
  <XYPosition x="200.0" y="50.0"/>
  <Depth value="12.0"/>
  <SetPower>
     <AllPropellers value="75.0"/>
  </SetPower>
</Waypoint>
```

Phoenix Behavior Sequence:

```
DEPTH    39.525612
RPM      525
WAYPOINT 658.7602 164.69005
```

Figure 5.2. An AVCL Waypoint Behavior and an Equivalent Phoenix UUV Behavior Sequence Automatically Generated from an XSLT Stylesheet

The solution to this potential shortcoming is the use of XSLT extensions that provide access to methods and programs written in other languages that implement

functionality not directly available in XSLT. Computation of Cartesian coordinates within the AVCL processing stylesheets developed during this research, for instance, relies on Equations 5.1 and 5.2 where R is the equatorial radius of the earth in meters and Lat_0 / Lon_0 is the geographic position of the Cartesian coordinate frame origin. These functions are implemented as static Java methods that are accessed from within the XSLT stylesheets as required. Alternatively, the same extension mechanism might be utilized to access Geographical Information System functionality for increased accuracy or advanced geographic processing.

$$x = \frac{2\pi R\left(Lat - Lat_0\right)}{360} \qquad \text{(Eq. 5.1)}$$

$$y = \frac{2\pi\left(R\cos\left(Lat\right)\right)\left(Lon - Lon_0\right)}{360} \qquad \text{(Eq. 5.2)}$$

To summarize, XSLT stylesheet templates can typically map instances of AVCL task-level behaviors to a single Phoenix behavior. In some cases, more than one Phoenix behavior is utilized to accurately capture the semantics of an AVCL task-level behavior. Regardless of the number of behaviors required, XSLT stylesheet templates that process AVCL task-level behaviors are self contained in that they neither rely on the outcome of previously invoked templates nor influence the execution of subsequently activated templates.

c. Conversion of AVCL for the ARIES UUV

In many respects ARIES UUV waypoint lists are the simplest target data format for AVCL task-level behavior scripts. After all, they consist of little more than a series of identically formatted numerical sequences. In fact the AVCL Waypoint behavior template is the only one in the AVCL-to-ARIES stylesheet that actually generates output. The templates for all other behaviors simply update commanded control parameters.

There is, however, a significant difficulty imposed by the self-contained nature of ARIES waypoints—XSLT's lack of side effects (particularly the immutability of XSLT variables) might seem to preclude the use of templates to update command parameters for later use. The common iterative pattern used in XSLT stylesheets does not retain information from iterative step to iterative step. If a series of AVCL task-level

behaviors is processed using normal XSLT iteration, control parameter information from one behavior is not retained for use by templates processing subsequent behaviors. If a MakeDepth behavior is encountered, the commanded depth is not ordinarily retained outside of the original MakeDepth XSLT template and is therefore not available when a Waypoint behavior is encountered. In this case the generated waypoint command will not accurately reflect the intent of the AVCL task-level behavior script.

This difficulty was overcome through the development of an XSLT programming pattern that achieves the functionality of mutable variables using XSLT template parameters. Parameter values are updated as a document is processed by explicitly controlling iteration rather than utilizing XSLT's default iteration model. Described in pseudocode in Figure 5.3, the simulated mutable variable algorithm intentionally applies templates one at a time. This differs significantly from the more common pattern which relies on the XSLT processing engine to apply templates to multiple elements matching a single XPath expression. At the top level, the stylesheet applies the appropriate template for the first task-level behavior. All potentially mutable variables are included in the initial template application as parameters with default values. Templates then instantiate internally immutable variables corresponding to each parameter and assign values to variables based on the content of the behavior element being processed. Variables corresponding to parameters whose value is updated by the task-level behavior being processed are set to the new value. For all others the existing parameter value is used. After generating any required output, the template invokes the appropriate template for the current element's next sibling. The variables instantiated by the current template are used as the parameters to the next template. In this way task-level behaviors are processed in order and information is maintained and updated along the way.

Using this new mutable-variable pattern, the templates for AVCL behaviors such as MakeDepth, MakeAltitude, ObtainGPS, SetPower and MakeSpeed update their respective command parameter without affecting others. In the AVCL-to-ARIES translation stylesheet, seven of the 11 ARIES waypoint fields listed in Table 5.2 are directly or indirectly maintained in this manner. Exceptions are the Cartesian coordinates that are specified in the Waypoint behavior, and the waypoint time out and

GPS popup duration which are computed during translation based on the current vehicle state. The most recently ordered horizontal position is maintained as well, so that the waypoint time out values can be automatically generated and the actual location of positions described in relative terms can be determined.

```
begin XSLT processing
     variable B1 = the first task-level-behavior
     apply template for B1
          with default parameters d1 to dn
end XSLT processing

begin template for task-level-behavior Bi
     with parameters p1 to pn
     for k = 1 to n
          variable vk
               if Bi updates pk
                    vk = new_pk
               else
                    vk = pk
     generate required output for Bi
     apply template for Bi+1
          with parameters v1 to vn
end template
```

Figure 5.3. Algorithm for Achieving Mutable Variables in XSLT using Template Parameters and Explicitly Controlled Iteration

Also worth noting is the use of XSLT to detect mission programming errors and mission-vehicle incompatibilities. Despite the intended application of AVCL to arbitrary vehicles, it is not difficult to define an AVCL mission that cannot be executed on a specific vehicle. The ARIES, for instance, is not hover-capable, so AVCL task-level behavior scripts that include Hover behaviors are not compatible with the ARIES. The Hover template in the AVCL-to-ARIES stylesheet is used to identify the incompatibility and notify the operator so that corrections can be made before the script is used with an actual vehicle. Other common incompatibilities include the attempted use of open-loop or closed-loop / open-ended behaviors. In some cases errors of this sort are not only detectable but may also be correctable by the stylesheet through the use of dead reckoning to convert closed-loop / open-ended behaviors to waypoints.

Despite the apparent simplicity of ARIES waypoint lists, their generation from AVCL task-level behavior scripts using XSLT is not as straightforward as it might initially appear. The immutability of XSLT variables and the existence of AVCL behaviors that are not compatible with the ARIES vehicle are both issues that must be dealt with. These issues, however do not inhibit the applicability of the proposed common data model to express ARIES waypoint lists. Ultimately, the correlations shown in Figure 5.4 are used to map content from AVCL task-level behaviors to the ARIES UUV waypoint fields. Solid lines indicate task-level behavior content that is mapped for every instance of that particular behavior—that is, content that is always present and therefore always used in the next generated waypoint. Dashed lines, on the other hand, indicate mappings that apply only if the relevant optional content is present in the task-level behavior.

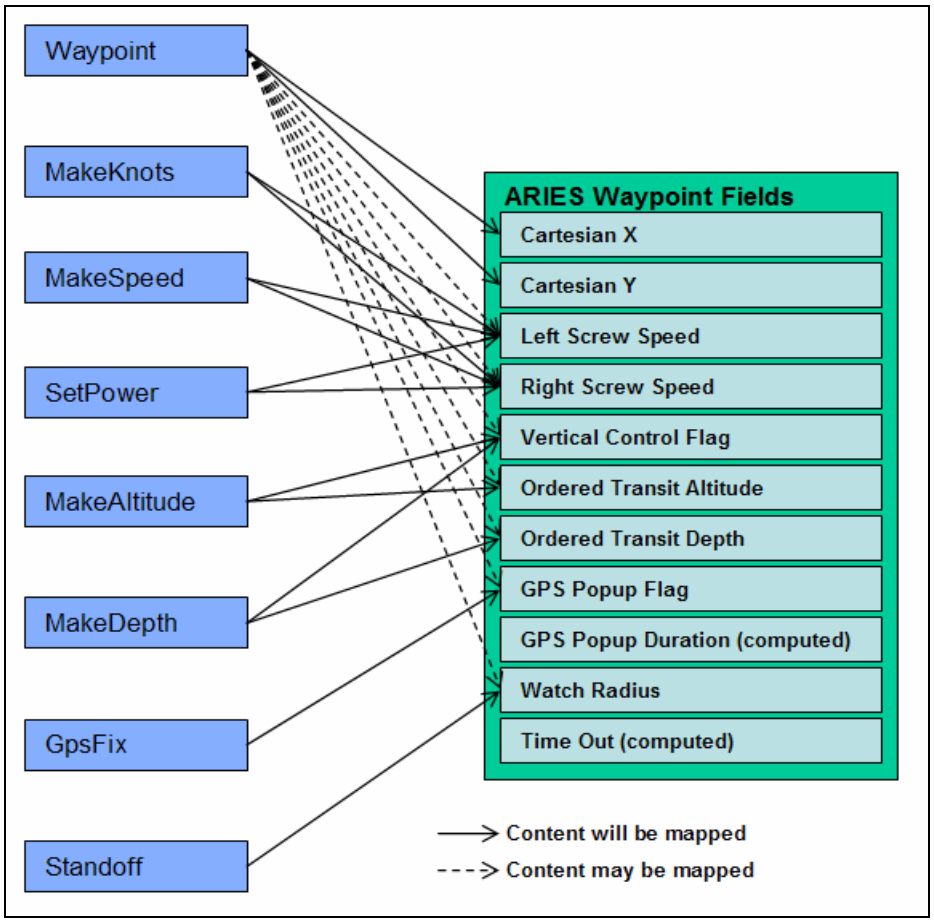


Figure 5.4. Data Mappings from AVCL Task-Level Behaviors to ARIES UUV Waypoint Fields

d. Conversion of AVCL for the Seahorse UUV

Like ARIES missions, Seahorse UUV missions take the form of a command script wherein each task is completely self contained. This implies that many of the translation issues and their means of resolution are similar to those of ARIES. In particular, the simulation of mutable variables along with the detection (and possible correction) of mission programming errors described in the preceding section are both used extensively in the AVCL-to-Seahorse translation stylesheet. Additionally since Seahorse uses latitude and longitude to specify absolute positions and Cartesian coordinates for relative positions the previously discussed XSLT extension mechanism is used to implement Equations 5.3 and 5.4 for the purpose of converting AVCL Cartesian positions for use by Seahorse.

$$Lat = \frac{360x}{2\pi R} + Lat_0 \qquad \text{(Eq. 5.3)}$$

$$Lon = \frac{360y}{2\pi R \cos(Lat)} + Lon_0 \qquad \text{(Eq. 5.4)}$$

This is not to say that XSLT stylesheets which translate AVCL for the Phoenix and ARIES vehicles encountered and resolved all issues required for the successful translation of AVCL for the Seahorse. On the contrary, the Seahorse scripting language poses several unique challenges. In particular, despite the fact that ARIES commands are inherently different than AVCL task-level behaviors, they are designed to control similar aspects of vehicle action in a similar fashion. Seahorse tasks, on the other hand, do not necessarily operate in the same way, often packaging implied vehicle actions differently. The GPS Fix order, for example, has a Return to Starting Point parameter that can be used to direct the vehicle to return to the previous waypoint upon obtaining the fix. In some cases Seahorse tasks imply vehicle actions that do not directly correspond to AVCL task-level behaviors (e.g., surface for the specific purpose of communicating). Other tasks include implicit data-gathering instructions (e.g., the Collect SVP parameter of the Surface Comms order that can direct the vehicle to collect sound velocity profile data while surfacing). All of these situations must be accounted for in a stylesheet intended to generate Seahorse scripting language.

In some cases a default value can be used (e.g., it might be acceptable to always order the vehicle to collect the sound speed profile while surfacing). In others the organization of the AVCL task-level behavior script can be used to implicitly handle the situation in an acceptable way. Figures 5.5 and 5.6 provide an example resolution of this issue. Figure 5.5 depicts an AVCL behavior sequence ordering the vehicle to transit to a waypoint, surface for one minute and return to the previous waypoint. The most concise way for a Seahorse script to express this is to use a Waypoint Navigation order followed by a Surface Comms order with the Return to Starting Point parameter set to true. However, generation of this sequence from the XSLT stylesheet requires an explicit check to see if the MakeDepth and Wait behaviors are immediately followed by a Waypoint returning the vehicle to the previous point. It is simpler to set the Return to Starting Point parameter to false for all Surface Comms tasks and process the subsequent Waypoint separately. This results in the Seahorse task sequence depicted in Figure 5.6 which commands the desired behavior. This pattern allows for a simpler stylesheet since it can effectively ignore issues that might otherwise require significant special handling.

```
<Waypoint>
  <LatitudeLongitude
    latitude="30"
    longitude="-118"/>
  <Depth value="12"/>
  <SetPower>
     <AllPropellers value="75"/>
  </SetPower>
</Waypoint>
<MakeDepth value="0"/>
<Wait value="60"/>
<Waypoint>
  <LatitudeLongitude
    latitude="30"
    longitude="-118"/>
  <Depth value="12"/>
</Waypoint>
```

Figure 5.5. An AVCL Task-Level Behavior Sequence Ordering a UUV to Proceed to a Waypoint, Surface, and Return to the Previous Waypoint and Depth

```
Start_Order               : Waypoint_Navigation_Order
Scheduling_Info_Is_Timed  : False
Destination_Latitude      : 30 Degrees
Destination_Longitude     : -118 Degrees
Transit_Mode              : Steer_to_Line
Transit_Depth             : 12 Meters
Transit_Speed_In_Water    : 5.25
Use_SSS                   : True

Start_Order               : Surface_Comms_Order
Scheduling_Info_Is_Timed  : False
Collect_SVP               : True
Return_to_Depth           : False
Return_to_Starting_Point  : False
Take_GPS_Fix              : False
Perform_RF_Comms          : True

Start_Order               : Waypoint_Navigation_Order
Scheduling_Info_Is_Timed  : False
Destination_Latitude      : 30 Degrees
Destination_Longitude     : -118 Degrees
Transit_Mode              : Steer_to_Line
Transit_Depth             : 12 Meters
Transit_Speed_In_Water    : 5.25
Use_SSS                   : True
```

Figure 5.6. An XSLT-Generated Seahorse UUV Task Sequence Equivalent to the Task-Level Behavior Sequence of Figure 5.5

Unfortunately, not every Seahorse task construct can be dealt with implicitly or through the use of defaults, so it is often necessary to perform tests on the neighbor, ancestor and descendant elements in order to resolve the output requirements of the current element. It is often possible to infer the proper interpretation of an element from the neighboring task-level behaviors as is the case for the Waypoint behavior. Since the last task in a Seahorse mission must be the rendezvous order, if a Waypoint behavior has no following Waypoint siblings, it is translated as a rendezvous order. Otherwise it is translated as a Waypoint Navigation order. In a similar fashion, the siblings of a MakeDepth behavior with a value of zero can be checked to determine whether it is to be interpreted as a Surface Comms order (if the behavior is immediately

followed by a Wait, WaitUntilTime or SendMessage behavior), whether a GPS Fix is to be taken while on the surface (if there are no sibling behaviors that will direct the vehicle to submerge between the MakeDepth behavior and a GpsFix behavior), or whether a rendezvous order is to require a GPS fix at the destination (when the last Waypoint behavior in the AVCL task-level behavior script has a following GpsFix behavior).

A few aspects of some Seahorse orders cannot be inferred from other behaviors in the AVCL script, so more interpretation is required to accurately translate AVCL behavior scripts for the Seahorse. For example, the Surface Comms order can either allow or disallow the use of radio frequency communications. While it may make sense to always allow radio communications, this comes at the expense of sacrificing vehicle capabilities. The same can be said of the ability to order the vehicle to collect sound velocity profile data while surfacing.

In order to deal with data format elements that are truly vehicle-specific, the AVCL task-level behavior set includes a MetaCommand behavior. This behavior has no direct effect on any vehicle control parameters, so it can often be ignored during translations. It does, however, provide a means of capturing vehicle-specific information that is not representable in other AVCL structures. XSLT stylesheets translating AVCL check the name and content attributes of MetaCommand behaviors to determine if they relate to the stylesheet's target language and adjust their processing accordingly. For instance, a MetaCommand behavior with a name attribute value of "obtainSVP" tells the AVCL-to-Seahorse stylesheet how to set the Collect SVP parameter of the next Surface Comms order (based on the value of the MetaCommand behavior's content attribute). Stylesheets targeted to vehicles other than Seahorse will ignore the MetaCommand behavior. If the stylesheet does not encounter the MetaCommand behaviors relating to its target data format before generating the content to which they apply, the stylesheet can still use default values ("true" for both the Perform Radio Frequency Comms and Collect SVP parameters of the Seahorse Surface Comms order). The MetaCommand name attribute values that the AVCL-to-Seahorse stylesheet supports and their corresponding meanings are listed in Table 5.6.

Name Attribute Value	Possible Content Attribute Values	Description
useSSS	true or false	The value of the next Waypoint Navigation order's Use SSS parameter.
rfComms	true or false	The value of the next Surface Comms order's Perform RF Comms parameter.
obtainSVP	true or false	The value of the next Surface Comms or GPS Fix order's Collect SVP parameter.
rendezvous	[no value]	Indication that the next Waypoint behavior is to be interpreted as the last one.

Table 5.6. AVCL MetaCommand Name Attribute Values Used by the XSLT Stylesheet Targeted to the Seahorse UUV Tasking Language

Data mappings used in the translation of AVCL task-level behaviors to Seahorse UUV orders are depicted in Figures 5.7 through 5.9. As with Figure 5.4, solid lines represent mappings from required task-level behavior content to the target Seahorse orders while dashed lines indicate mappings of optional content. Since a large variety of task-level behavior content is potentially mappable to multiple Seahorse tasks, the actual mapping utilized for a particular task-level behavior depends upon the type of Seahorse order that is required. In many cases data is saved for use in the next generated Seahorse order. In other cases the context of the particular behavior (i.e., the types and content of neighboring behaviors) dictates the immediate generation of a particular Seahorse order. Regardless, the most current data is always used for any Seahorse order that is generated.

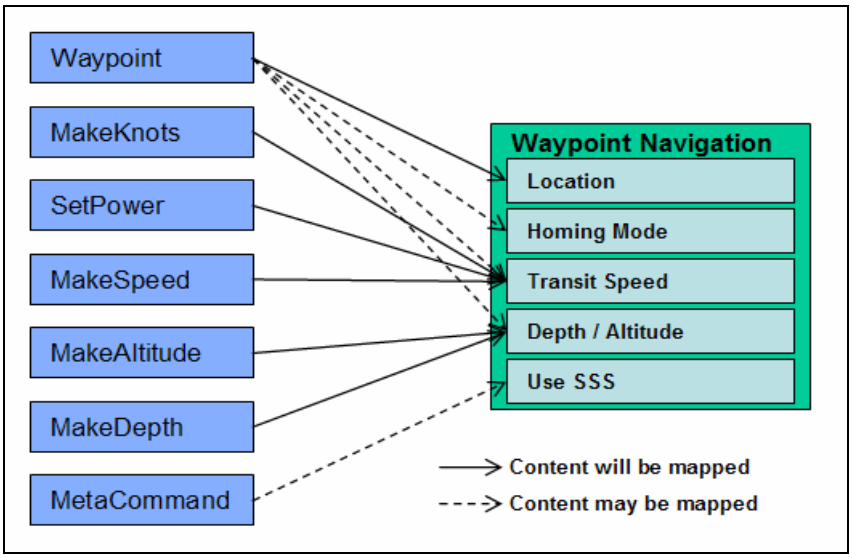


Figure 5.7. Data Mappings from AVCL Task-Level Behaviors to the Seahorse UUV Waypoint Order

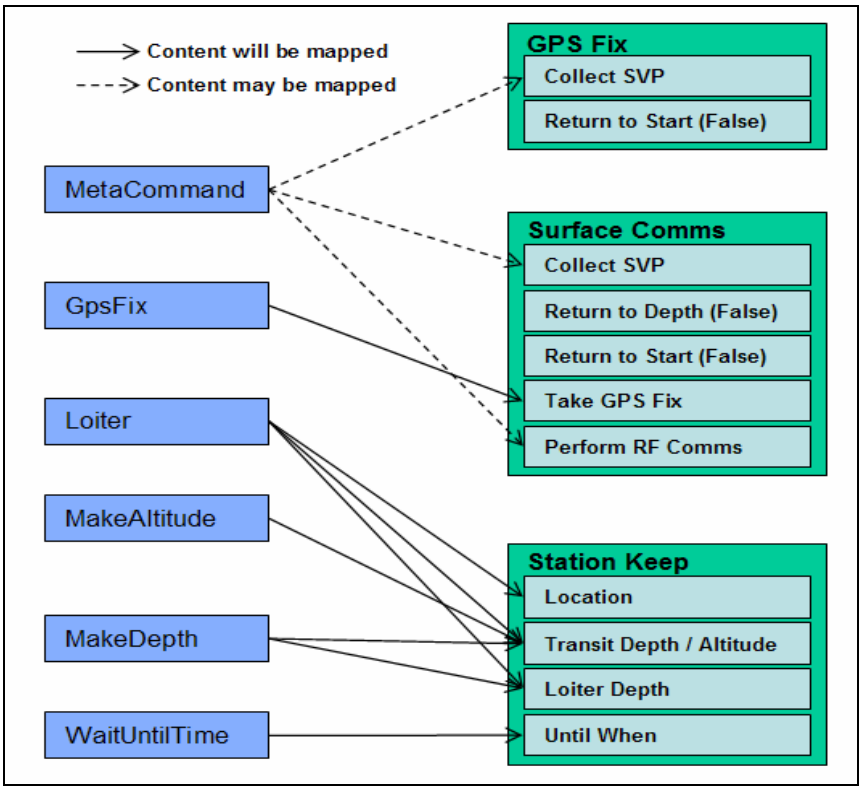


Figure 5.8. Mappings from AVCL Task-Level Behaviors to Seahorse GPS Fix, Surface Comms and Station Keep Orders

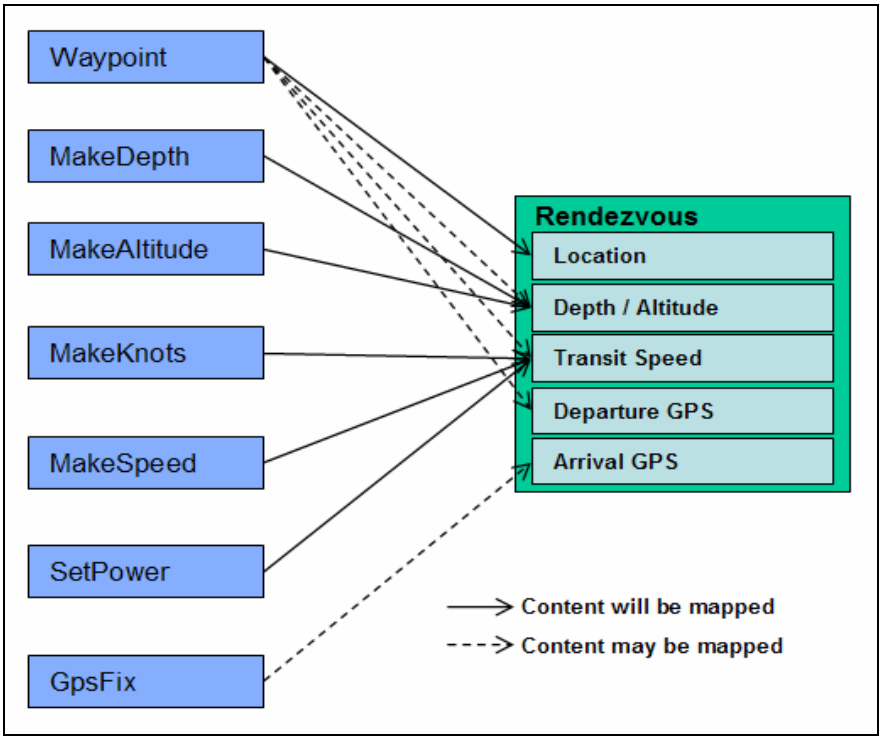


Figure 5.9. Mappings from AVCL Task-Level Behaviors to the Seahorse Rendezvous Order

The syntax of the Seahorse UUV tasking language is significantly more complex than those of the Phoenix and ARIES vehicles. Nevertheless, an XSLT stylesheet can be designed to translate AVCL task-level behavior scripts for the Seahorse UUV. The implementation of more robust content checks within the stylesheet and the use of the AVCL MetaCommand to capture information specific to the Seahorse vehicle facilitates accurate translation despite the increased complexity of the target language.

e. Conversion of AVCL for the REMUS UUV

Despite its increased complexity, the REMUS command language does not pose any specific translation difficulties whose ultimate resolution method has not already been discussed. In fact this is not surprising since the REMUS command language is a task scripting language with identical execution semantics to ARIES waypoint lists and Seahorse task scripts (i.e., tasks are executed in order and do not commence until the preceding task completes). Additionally, many of the 13 available REMUS objectives are similar enough to allow their generation by the same XSLT stylesheet templates (e.g., there are three REMUS waypoint objectives, and four mission-start objectives). The translation of AVCL task-level behavior scripts for the REMUS vehicle differs only by the increased use of the XSLT mutable-variable pattern, behavior-context checking, and AVCL MetaCommand behaviors.

MetaCommand behaviors, in particular, play an increased role in the AVCL-to-REMUS stylesheet and is summarized in Table 5.7. As with the AVCL-to-Seahorse stylesheet, MetaCommand behaviors are used to indicate the values of vehicle-specific REMUS command parameters. Unlike the Seahorse command language, however, the REMUS language contains not only vehicle-specific objective parameters, but vehicle-specific objectives as well. There is, for example, no AVCL task-level behavior that can capture the semantics of the REMUS Wait Prop objective. Five of the 13 objective types of Table 5.7 do not translate meaningfully into AVCL task-level behaviors, but it is a natural extension of the previous MetaCommand use to indicate the intended inclusion of these objectives to the XSLT stylesheet processor with MetaCommand behaviors.

Name Attribute Value	Possible Content Attribute Values	Description
calibrateDepth	yes or no	Value for the launch command "Auto calibrate depth sensor" parameter.
clearCalibration	yes or no	Value of the next compass calibration command's "Clear calibration?" parameter.
compassCalibration	yes or no	Calls for a compass calibration command at the current position in the script.
depth	positive float	Transponder depth for a transponder location.
hardware	String	Location type and label for a transponder location.
latitude	float	Latitude of a transponder.
longitude	float	Longitude of a transponder.
moveAway	positive float	Value of the next compass calibration command "Move away duration (seconds)" parameter
sidescanRange	same, 5, 10, 20, 30, 40, 50 or 75	Value of the next Navigate, Dead Reckon or Transponder Home command "Sidescan Range" parameter.
trackPingInterval	positive float	Value of the next Navigate, Dead Reckon or Transponder Home command "Track ping interval (seconds)" parameter.
transponderLabel	String	Transponder location label for the next Transponder Home command.
triangleAltitude	positive float	Value to use for the next triangle altitude control "Triangle altitude" parameter.
triangleMinimum	positive float	Value to use for the next triangle depth or altitude control "Triangle minimum (m)" parameter.
triangleMaximum	positive float	Value to use for the next triangle depth or altitude control "Triangle maximum (m)" parameter.
triangleRate	positive float	Value to use for the next triangle depth or altitude control "Triangle rate (m/min)" parameter.
verifyTransponderRange	yes or no	Value for the launch command "Verify range to nearest transponder" parameter.
waitDepth	positive float	Calls for a Wait Depth command at the current script location (should follow set position).
waitMagnet	none	Calls for a Wait Magnet command at the current script location (should follow set position).
waitProp	positive integer	Calls for a Wait Prop command at the current script location (should follow set position).
waitRun	none	Calls for a Wait Run command at the current script location (should follow set position).
waypointNavMode	best, deadReckon or transponderHome	Directs the type of waypoint command to use.

Table 5.7. AVCL MetaCommand Name Attribute Values Used by the XSLT Stylesheet Targeted to the REMUS UUV Family

MetaCommand behaviors are also used to indicate desired vehicle-specific control modes. Given their other applications, the use of a MetaCommand behavior to note the intended use of a Navigate objective rather than a Dead Reckon or Transponder Home objective, for example, seems reasonable. Additionally, REMUS-specific triangle vertical-control modes (which direct the vehicle to vary depth or altitude between upper and lower bounds) are captured by MetaCommand behaviors where the name attribute correlates to the control mode parameter (e.g., triangle minimum) and the content attribute is set to the desired parameter value.

A final use of MetaCommand behaviors to capture REMUS-specific information is the specification of the locations of the transponders that the REMUS vehicles utilize for position tracking. Each transponder location is encoded with four MetaCommand behaviors—one for the type and label, and one each for the latitude, longitude and depth. These particular MetaCommand behaviors are parsed by the stylesheet before any REMUS objectives are generated in order to construct the mission file's locations section. These are the only locations required when generating REMUS missions from AVCL task-level behavior scripts since all other geographic positions within the script are included in their respective commands.

As with the Seahorse-specific MetaCommand behaviors, those relating to REMUS objectives can be effectively ignored by stylesheets targeted to vehicles outside of the REMUS UUV family. Missions can therefore be defined from the start with multi-vehicle compatibility in mind even when vehicle-specific data must be generated during the translation to one or more target vehicles.

To summarize, AVCL MetaCommand behaviors are used to encode REMUS-specific objective parameter values, commands and navigation transponder locations, and also to differentiate between the available REMUS waypoint types. This extensive use of MetaCommand behaviors does not come without overhead, most frequently in the form of additional mutable variable requirements. Of the 18 mutable variables used to implement the AVCL-to-REMUS stylesheet, nine maintain information that is inferred from MetaCommand content. Fortunately, since all mutable variables are

handled simultaneously (i.e., the algorithm is not reimplemented for each variable), additional mutable variables do not add significantly to the overall stylesheet complexity.

Mappings used to generate REMUS missions from AVCL task-level behavior scripts are depicted in Figures 5.10 through 5.12. The commands not presented are generated entirely from MetaCommand content (or default values if the expected MetaCommand behaviors are not encountered before the objective is to be generated) with the exception of the Navigate Rows objective, which is never generated by the AVCL-to-REMUS stylesheet since potentially applicable AVCL CompositeWaypoint behaviors are converted to a series of individual Waypoint behaviors prior to translation.

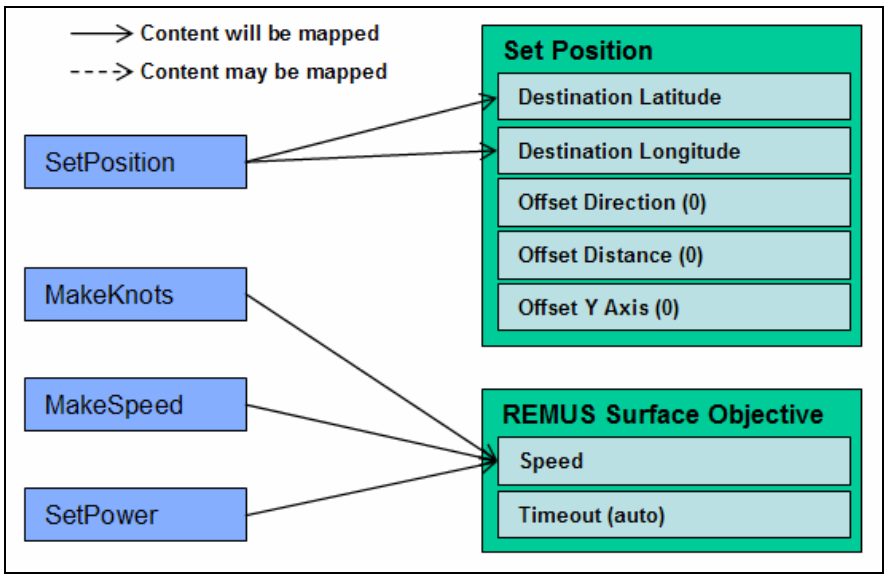


Figure 5.10. AVCL Data Mapping to the REMUS UUV Set-Position and Surface Objectives

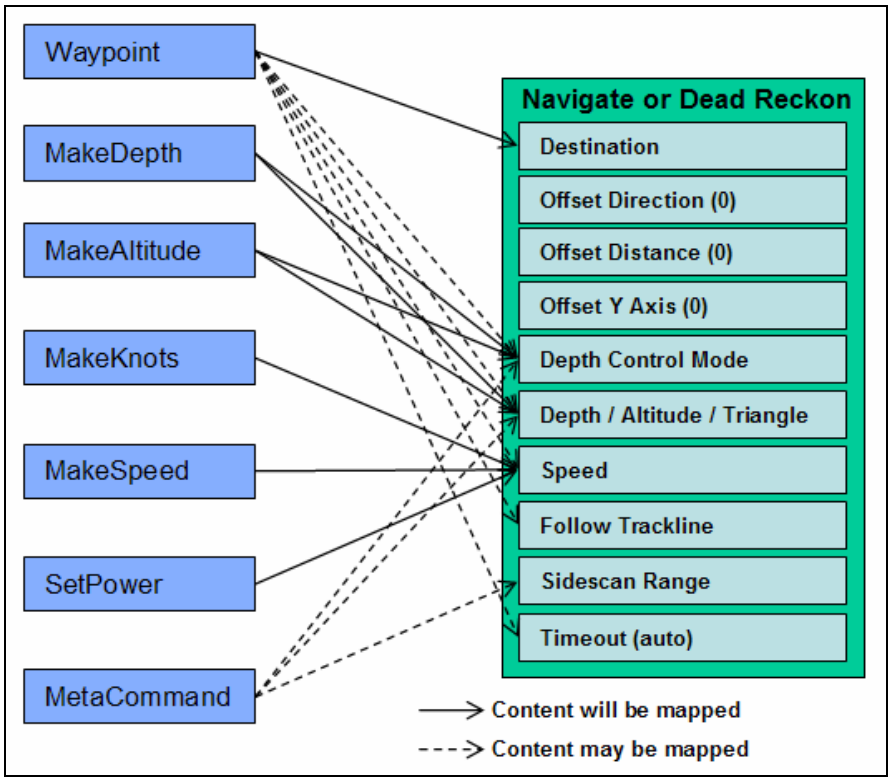


Figure 5.11. AVCL Data Mapping to the REMUS UUV Navigate and Dead-Reckon Objectives

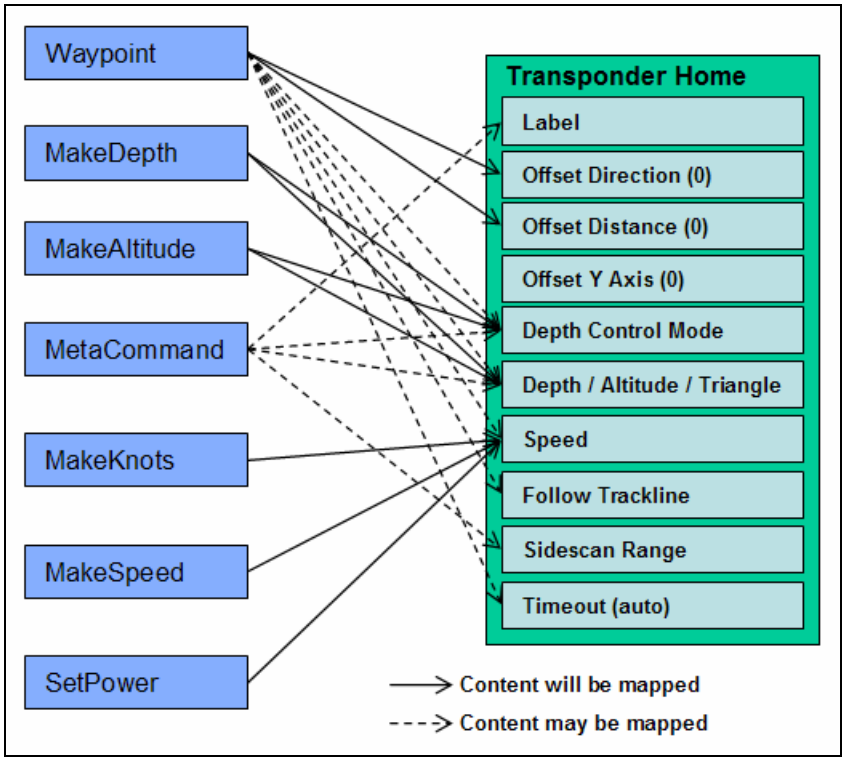


Figure 5.12. AVCL Data Mapping to the REMUS UUV Transponder-Home Objective

2. Generation of Data-Model-Compliant XML from Vehicle-Specific Text Documents

a. Context-Free-Grammar-Based Translation

The preceding discussion illustrates the utility of XSLT in translating data-model-compliant XML to arbitrary text-based data formats used by autonomous vehicles. This is, however, only half the problem. In order to use a common data model as a bridge between vehicle-specific data formats, it is necessary not only to translate data-model-compliant documents for specific vehicles, but to convert vehicle-specific data into data-model-compliant documents. Since vehicle-specific data formats are generally not XML-based, an XSLT stylesheet cannot be used for the conversion of vehicle-specific data to common data-model-compliant XML. There is, in fact, no generally recognized tool of choice suitable for this purpose. A simple observation, however, can be made that vehicle-specific data formats do impose rigorously defined lexical, semantic, and structural constraints even though they are not XML-based. This observation provides the basis for a methodology for automated parsing and conversion of vehicle-specific data to data-model-compliant XML.

More formally, each vehicle-specific data format is actually a context-free language. Mathematically speaking a context-free language is the set of strings derivable from a context-free grammar (Hopcroft, et al., 01). The implication of the previous statements is that there exists a context-free grammar corresponding to any vehicle-specific data format in which we might be interested. It stands to reason that the context-free grammar can be used to both generate and parse instances of the vehicle-specific format.

A context-free grammar is formally specified with four components: V, T, P and S, where V is a set of variables, T the set of terminal symbols in the context-free language, P a set of production rules, and S the set of available start symbols (a non-empty subset of V) (Hopcroft, et al., 01). As a simple example, consider the productions $P \rightarrow (\ )$ and $P \rightarrow (P)$ where P is a variable, '(' and ')' are terminal symbols and the production symbol ($\rightarrow$) means that the variable on the left can be expanded into the sequence of terminal symbols and variables on the right. These particular productions are capable of generating all strings of balanced parentheses (i.e., strings of the form

"((()))"). An appropriate context-free grammar for the context-free language consisting of all balanced parentheses strings can therefore be fully defined using Equation 5.5. Simplicity of the example notwithstanding, context-free grammars can be a powerful tool for recursively defining a variety of complex languages. In fact context-free grammar production rules of the form described above provide the basic building blocks of the XML DTD and can be used to define high-level programming languages as well (Hopcroft, et al., 01).

$$G = \left(\{P\}, \{(,)\}, \{P \to (\ ),\ P \to (P)\}, \{P\}\right) \qquad \text{(Eq. 5.5)}$$

The use of context-free grammars to support the translation of vehicle-specific data into AVCL consists of three steps: definition of a Chomsky-Normal-Form grammar corresponding to the vehicle-specific data format, use of the Cocke-Younger-Kasami algorithm to generate a parse tree corresponding to the vehicle-specific data, and the conduct a depth-first parse-tree traversal to convert it to an equivalent AVCL document (Davis, 05).

The productions of a Chomsky Normal Form context-free grammar have three significant characteristics:

- There are no useless symbols (i.e., variables or terminal symbols that do not appear in any terminal-string derivation beginning with the start symbol).
- There are no ε (null) productions (i.e., those of the form $A \to \varepsilon$).
- All productions are of the form $A \to BC$ or $A \to a$ where *A*, *B* and *C* are variables and *a* is a terminal symbol.

It can be proven that for any context-free language not containing the empty string (ε), there exists a Chomsky Normal Form context-free grammar capable of generating that context-free language (Hopcroft, et al., 01).

From a practical standpoint, defining a context-free grammar to generate a particular vehicle-specific data format involves determining the terminal symbols and writing the productions. The set of terminal symbols can consist solely of numbers as is the case with ARIES waypoint lists or can include keywords, symbols and numbers as in the Phoenix, Seahorse and REMUS command languages. Production definition can be fairly arbitrary. Nevertheless, the production rules ultimately determine the structure of

the parse tree, so it is advisable to utilize rules that have intuitive meaning. The production rules of Figure 5.13, for instance, can be used to derive the parse tree of Figure 5.14 corresponding to a single Phoenix waypoint behavior. Ultimately, defining meaningful production rules facilitates the final step of the translation process. Defining production rules in an intuitive manner also allows the reuse of variables throughout the set of production rules (e.g., the Position2D and Position3D variables can be used in any production using a two dimensional or three-dimensional position respectively).

```
Command → WaypointToken + Position3D
Position3D → Position2D + Double
Position2D → Double + Double
WaypointToken → "WAYPOINT"
Double → any floating point number
```

Figure 5.13. Context-Free Grammar Production Rules for Generating a Phoenix UUV Waypoint Behavior

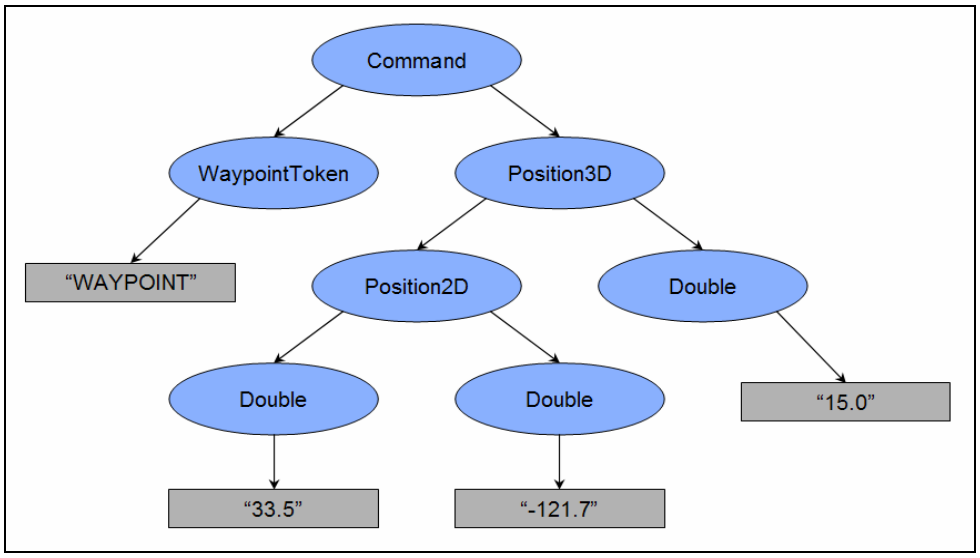


Figure 5.14. A Parse Tree Corresponding to a Single Phoenix UUV Waypoint Behavior Based on the Production Rules of Figure 5.13 (After: Davis, 05)

The second step in the translation process is the application of the Cocke-Younger-Kasami algorithm (Figure 5.15) to a context-free language instance. The Cocke-Younger-Kasami algorithm uses dynamic programming (Corman, et al., 90) to

parse context-free language instances in a bottom up fashion. The result of the algorithm is a binary parse tree representing a context-free grammar derivation of the context-free language instance (for the sake of simplicity, the algorithm as depicted in Figure 5.15 tests for context-free language membership but does not actually generate a parse tree). The Cocke-Younger-Kasami algorithm was chosen for parsing context-free language instances for two reasons. First it is among the more efficient context-free language parsing algorithms available, having a computational complexity of $O(n^3)$ where *n* is the length of the string being parsed (Corman, et al., 90). There are, in fact, algorithms available that are more efficient for certain context-free languages, but the Cocke-Younger-Kasami is a good choice if reasonable performance is required for all context-free languages. The second reason for using the Cocke-Younger-Kasami algorithm is that it is universally applicable. That is, it can be used to parse an instance of any Chomsky Normal Form context-free grammar. This relative language-independence is one of the most significant strengths of this translation approach.

```
Let the input string be a sequence of n letters a1…an
Let V1…Vr be the set of CFG symbols (V)
Let S be the set of indices of V corresponding to
   context-free grammar start symbols
Let P[n, n, r] be a Boolean array initialized to false

For i = 1 to n
    For each unit production Vj → ai
        P[i, 1, j] = true

For i = 2 to n – Length of span
    For j = 1 to n – i + 1 – Start of span
        For k = 1 to i – 1 – Partition of span
            For each production VA → VBVC
                if P[j, k, B] == true and
                   P[j + k, i – k, C] == true
                    then P[j, i, A] = true

if P[1, n, x] is true (x is an element of S)
    then the string is in the context-free language
    else the string is not in the context-free language
```

Figure 5.15. The Cocke-Younger-Kasami Algorithm for Parsing Chomsky-Normal-Form Context-Free Language Instances (After: Hopcroft, et al., 01)

The final step in the translation process is converting the parse-tree representation of a vehicle-specific document to AVCL. Conversion of a parse tree to AVCL relies on a depth-first traversal of the tree. As with XSLT processing of an XML document, templates defining what actions are to be taken are applied at each node. Actions in most cases call for recursive processing of the current node's left and right child, and in some cases include the generation of AVCL content.

As an example, upon arriving at the Command node of Figure 5.14, the applicable template calls for the generation of an AVCL Waypoint behavior corresponding to the Phoenix waypoint behavior. Since the structure of the parse tree is deterministic, it is unlikely that further recursive processing of the subtree will be required. All of the information required to generate the AVCL task-level behavior is in a known location relative to the current node and can be accessed directly. For instance the type of command is always specified by the left child of a Command node. Once the translator is aware that the current node contains a waypoint behavior, the content of the right subtree is known as well.

A noteworthy characteristic of the depth-first traversal is that the overall instance document is processed in order. For vehicle-specific data formats along the lines of those used in the ARIES and Seahorse UUVs, the corresponding AVCL is therefore generated in the same order. However, the sequence and structure of the AVCL corresponding to more complex data formats such as that of the REMUS family of vehicles may differ significantly from that of the vehicle-specific document. Since the parse-tree traversal is fixed, AVCL generation must be arbitrary (i.e., it must be possible to generate the AVCL document in any order). XML data binding facilitates this sort of document generation by creating the document as in-memory objects and deferring output serialization until tree construction is complete. Since the entire document is maintained in memory throughout the process, it is a fairly simple matter to access or modify existing content, add new content, and move or copy existing content from one location in the document to another.

Programmatic implementation of individual context-free grammars is accomplished through the definition of a dictionary class specific to the context-free

grammar that lexically maps tokens to their unary production rule, along with a context-free-grammar-specific class containing the binary production rules of the grammar. The implementation documented here is written in Java. In effect the dictionary and production classes perform the same function for vehicle-specific data formats that the XML schema does for AVCL—they formally constrain the structure and content of compliant documents and provide a means of automatically loading the document into a semantically meaningful programmatic data structure.

Implementation of the parse-tree translator is accomplished through the definition of a context-free-grammar-specific translator class. Just as the use of dictionary and production rule classes to constrain non-XML data is analogous to the use of an XML schema, the function of the translator class is analogous to that of the XSLT stylesheets discussed in the previous section. The parse-tree translators, however, perform their function in a different manner. XSLT stylesheets can traverse tree contained in the source document arbitrarily but must generate the output document serially. Conversely, the depth-first traversal of the parse tree by the context-free grammar translators uses only serial processing of the source document but allows arbitrary access to the output document.

To summarize, context-free grammars provide the basis for automated translation from arbitrary text-based, vehicle-specific data formats to common autonomous vehicle data-model-compliant XML. Analogous to the XSLT-based translations from AVCL to vehicle-specific formats, context-free-grammar-based translation of vehicle-specific formats to AVCL completes the translation chain between arbitrary text-based, vehicle-specific data formats.

b. Conversion of Phoenix UUV Command Files to AVCL

Of the four text-based, vehicle-specific data formats discussed in the previous section, the Phoenix behavior scripting language is the easiest to generate from AVCL task-level behavior scripts. The characteristics that make this so, namely the frequent one-to-one correlation between individual behaviors and the semantic similarity between the languages, also simplifies the translation of Phoenix behavior scripts to AVCL. In fact, the increased flexibility of AVCL in representing various parameter configurations enables a one to one mapping from Phoenix behaviors to AVCL task-level

behaviors in all but the most unusual cases. A summary of the Phoenix-to-AVCL translator mappings from the most commonly utilized Phoenix behaviors to AVCL is provided in Table 5.8.

Phoenix Behavior	AVCL Task-Level Behavior	Notes
Depth	MakeDepth	Units converted from feet to meters.
GPS	GpsFix	None
Heading	MakeHeading	None
Hover	Hover	Units converted from feet to meters. May include depth, heading and standoff.
Lateral	MoveLateral	None
Mission-Script	MissionScript	None
Planes	SetPlanes	None
Position	SetPosition	Units converted from feet to meters.
Quit	Quit	None
Rotate	MoveRotate	None
RPM	SetPower	RPM converted to percent of maximum.
Rudder	SetRudder	None
Thrusters-Off	Thrusters	Set the "enabled" attribute to "false."
Thrusters-On	Thrusters	Set the "enabled" attribute to "true."
Wait	Wait	None
WaitUntilTime	WaitUntilTime	None
Waypoint	Waypoint	Units converted from feet to meters. May include depth, speed and standoff.

Table 5.8. Mappings from Phoenix UUV behaviors to AVCL Task-Level Behaviors

The context-free grammar upon which the Phoenix-to-AVCL translator relies consists of 92 unary production rules, 39 binary production rules and 32 variables. The unary productions are capable of generating floating point numbers and white-space-free strings as well as the language's 90 behavior keywords. The unexpectedly large number of keywords arises from the availability of multiple keywords to specify the same behavior (e.g., a the Heading behavior can be ordered with a "heading," "course," or "yaw" keyword). There are a number of behaviors in the Phoenix behavior scripting language that are not directly representable in AVCL. By and large, these behaviors support mission and control testing in a virtual environment (Brutzman, 94). Since they do not affect vehicle control these behaviors are not included in AVCL although their functionality has been incorporated directly into the AUVW (Davis and Brutzman, 05).

If it is desirable to identify the presence of these behaviors in the generated AVCL document, they can be included as MetaCommand behaviors.

c. Conversion of ARIES UUV Command Files to AVCL

The most compact vehicle-specific context-free grammar developed for this research is the one corresponding to ARIES UUV waypoint lists. This is not surprising since all terminal symbols are numbers and every line of the waypoint list has the same format (with the exception of the first line of the file). A context-free grammar corresponding to the ARIES waypoint list context-free language, in fact, can be fully defined using only one unary production, 14 binary productions and 13 variables.

The actual translation of parse trees corresponding to ARIES waypoint lists into AVCL is also straightforward. In most cases, every element of an ARIES waypoint designation can be captured as a Single AVCL Waypoint. The only exceptions are waypoints that call for differential propeller settings, which require the individual propeller commands to be specified prior to the Waypoint command. Figure 5.16 shows the AVCL task-level behaviors corresponding to the translation of two ARIES waypoints. The first ARIES waypoint calls for both propellers to be powered at three volts (60 percent of maximum power) so the propeller setting is included in the generated Waypoint behavior. The second ARIES waypoint calls for the port propeller to be powered at 75 percent and the starboard propeller to be powered at 50 percent. In order to capture both commanded settings, they are placed immediately prior to (rather than within) the AVCL Waypoint behavior. Given the behavior-activation semantics of AVCL, this is equivalent to declaring the propeller orders within the Waypoint behavior.

d. Conversion of Seahorse UUV Command Files to AVCL

With 136 binary production rules, 46 unary production rules and 159 variables, the context-free grammar developed to support parsing of Seahorse mission files is the most complex one implemented in the course of this work. The complexity of this particular context-free grammar is brought about by the requirement to deal with parameters that may be optional, interchangeable or that may have more than one potential form. Nevertheless, use of the context-free grammar for translation of Seahorse mission files to AVCL is not negatively impacted.

```
<Waypoint>
 description="Aries absolute position waypoint">
  <XYPosition x="325.0" y="-25.0"/>
  <Depth value="25.0"/>
  <SetPower>
    <AllPropellers value="60.0"/>
  </SetPower>
  <ObtainGps value="true"/>
  <Standoff value="7.5"/>
  <TimeOut value="57.45"/>
</Waypoint>
<SetPower>
  <PortPropeller value="75.0"/>
</SetPower>
<SetPower>
  <StarboardPropeller value="50.0"/>
</SetPower>
<Waypoint
  description="Aries absolute position waypoint">
  <XYPosition x="425.0" y="-25.0"/>
  <Altitude value="15.0"/>
  <ObtainGps value="false"/>
  <Standoff value="10.0"/>
  <TimeOut value="66.6"/>
</Waypoint>
```

Figure 5.16. An AVCL Task-Level Behavior Sequence Corresponding to Two ARIES UUV Waypoints

Despite the increased complexity of the Seahorse context-free grammar, conversion of generated parse tree to AVCL is not difficult. Each task is contained within a single subtree, so once the subtree for a particular task is encountered it is a simple matter to gather the associated parameter values. Once the parameter values are collected AVCL task-level behaviors corresponding to the Seahorse order can be generated. Figures 5.17 through 5.21 show the specific AVCL behaviors that are generated for each Seahorse order, and from where their attribute values originate.

In general the sequence of AVCL task-level behaviors corresponding to a particular Seahorse order tends to be intuitive and the overall pattern is fairly consistent. MetaCommand behaviors (if required) are always placed first in the sequence. This

ensures that the content is available when subsequent behaviors are processed during transformation to another data format (e.g., the rfComms MetaCommand behavior is processed before the surfacing MakeDepth behavior). The AVCL behavior activation scheme dictates that depth and speed-related behaviors typically follow the MetaCommand behaviors since they potentially affect subsequent Waypoint, Loiter or Hover behaviors. Waypoint or Loiter behaviors follow the depth and speed behaviors. Finally, any post-order behaviors are generated. The most common behaviors falling into this category are those supporting the return-to-depth, return-to-start and post-arrival-GPS parameters of the Seahorse Surface Comms, GPS Fix and Rendezvous orders.

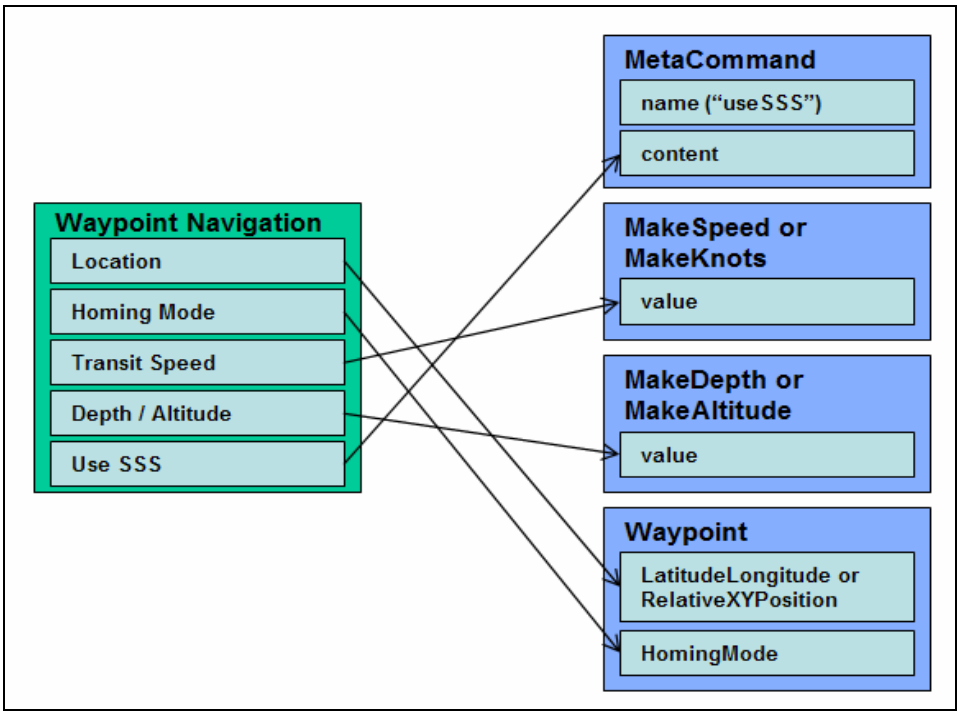


Figure 5.17. Data Mapping from a Seahorse UUV Waypoint Navigation Order to AVCL Task-Level Behaviors

Although similar, the mappings from Seahorse orders to AVCL task-level behaviors differ from the reverse mappings in one regard. Whereas AVCL content is often optional mapped only when present, all Seahorse order parameters are required, so the mapping always occurs. Depending on whether a particular parameter value equates to the AVCL default behavior, the corresponding task-level behaviors (or their content) may or may not be generated. Good examples of this are provided by the Return to

Depth and Return to Start parameters of the Surface Comms order (Figure 5.19). The task-level behaviors associated with these parameters are generated only if the value of the corresponding parameter is set to "true" since a setting of "false" is best represented by the absence of the corresponding task-level behavior. Optional behaviors are indicated in the figures by dashed borders.

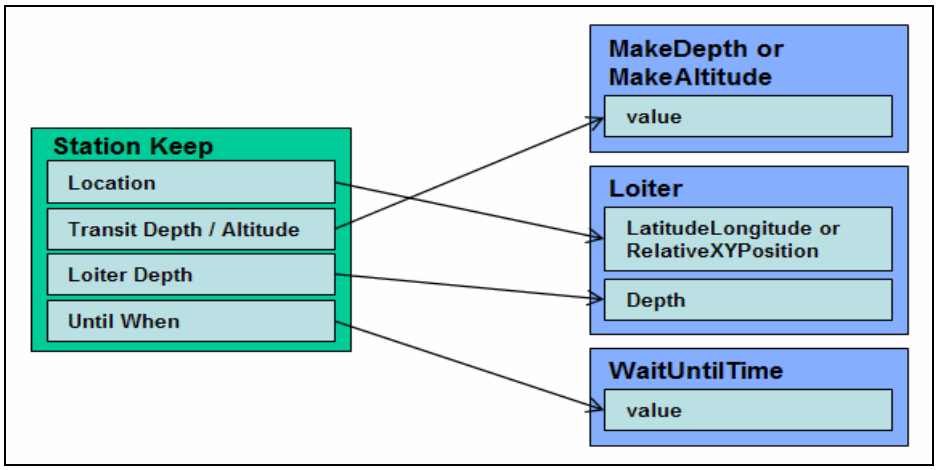


Figure 5.18. Data Mapping from a Seahorse UUV Station Keep Order to AVCL Task-Level Behaviors

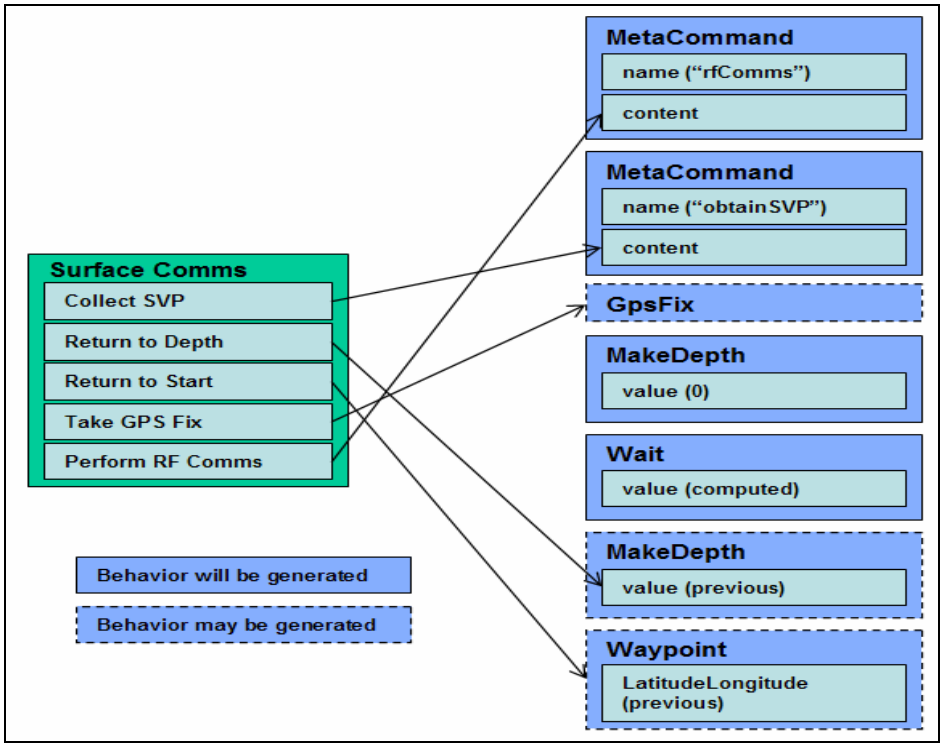


Figure 5.19. Data Mapping from a Seahorse UUV Surface Comms Order to AVCL Task-Level Behaviors

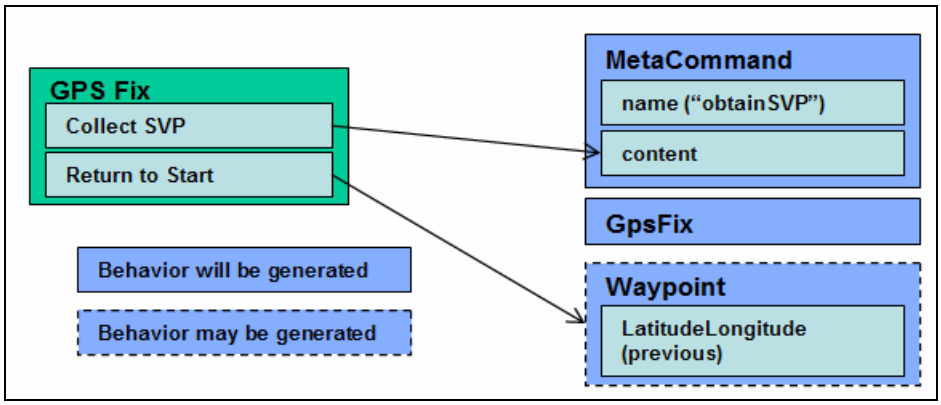


Figure 5.20. Data Mapping from a Seahorse UUV GPS Fix Order to AVCL Task-Level Behaviors

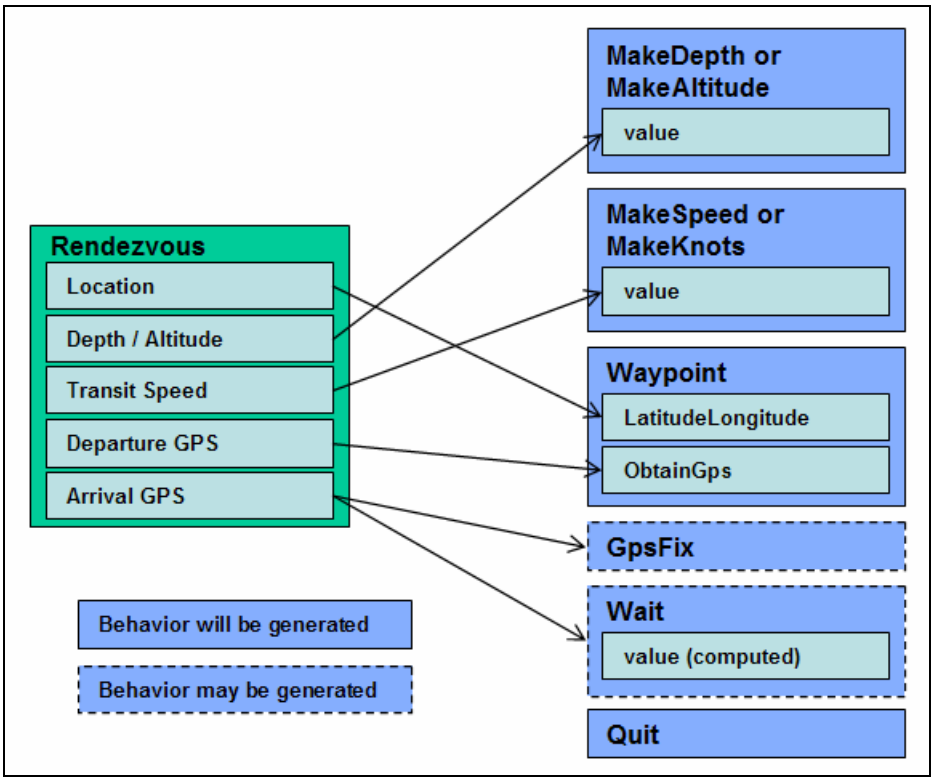


Figure 5.21. Data Mapping from a Seahorse UUV Rendezvous Order to AVCL Task-Level Behaviors

e. Conversion of REMUS UUV Command Files to AVCL

Significantly simpler than the context-free grammar designed for the Seahorse UUV, the context-free grammar implemented for the tasking language of the REMUS family of UUVs consists of 58 binary production rules, 119 unary production rules (as with Phoenix behaviors there are often multiple keywords with the same

meaning) and 52 variables. A noteworthy aspect of the parse trees generated using this context-free grammar is that the left subtree of the root node contains all of the locations (including those specifying transponder locations). The right subtree of the root node, on the other hand, contains all of the objectives. The depth-first parse tree traversal used during the translation process ensures that all locations are processed before any of the objectives. This is important for two reasons. First, it facilitates the placement of MetaCommand behaviors corresponding to the transponder locations at the beginning of the AVCL task-level behavior script, which in turn facilitates processing them first when transforming the task-level script back to a vehicle-specific format. Second, it ensures the processing of location descriptions before objectives that may reference them.

The use of a separate locations section in REMUS mission files and the use of position references and offsets introduce translator issues that were not encountered during the implementation of the Phoenix, ARIES or Seahorse translators. Relative positions and references, for instance, are included in the REMUS language in order to simplify programming through reuse of individual positions, however simplification is irrelevant to processing stylesheets and translators. For this reason all positions in a REMUS mission file (both those defined in the locations section and those defined in objectives) are converted to latitude and longitude—references and offsets are removed. Also required during location processing is the generation of a lookup table. As a location is processed, its type, label, latitude, longitude and transponder depth (if included) are stored in a hash table (using the label as the key) for use when processing objectives containing references to the position. Finally, the locations of navigation transponders must be incorporated into the AVCL task-level behavior script. The MetaCommand behaviors used for this purpose are generated as depicted in Figure 5.22.

After preprocessing the locations portion of the parse tree, translation of the objectives subtree into AVCL task-level behaviors is not significantly more difficult than translation of Seahorse orders. Of the 13 REMUS objective types, seven can be mapped to the single AVCL task-level behavior indicated in Table 5.9. Four of these map to MetaCommand behaviors that serve only to indicate the presence of the objective in the objective list and the values of any parameters. The remaining objective types are translated to AVCL task-level behaviors as depicted in Figures 5.22 through 5.24.

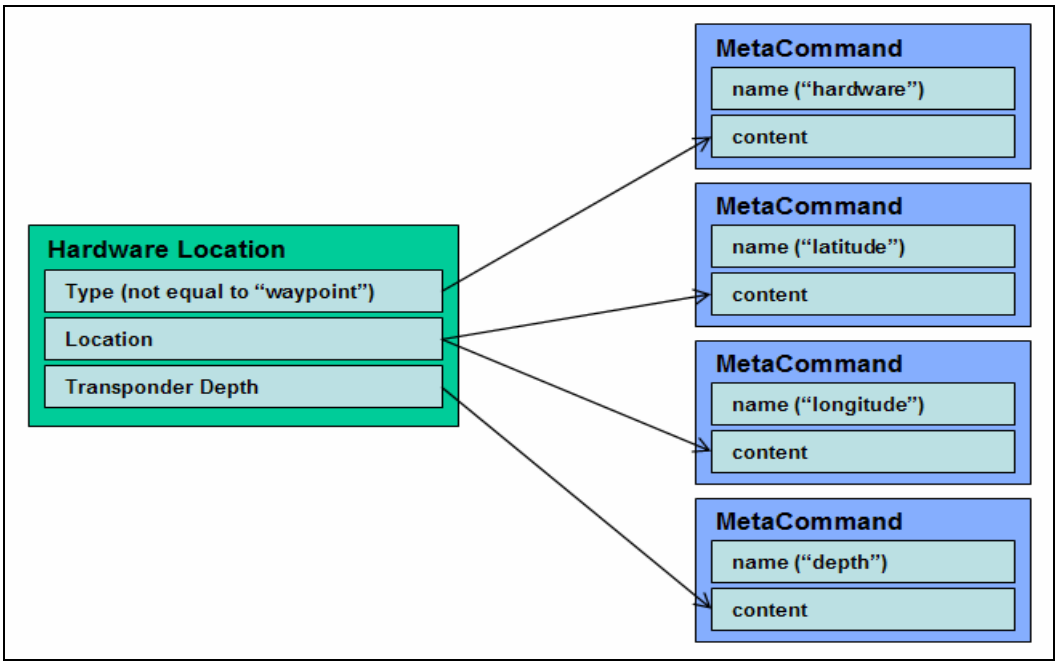


Figure 5.22. Data Mapping from a REMUS UUV Location Descriptor Defining the Position of a Navigation Transponder to AVCL MetaCommand Behaviors

REMUS Command	AVCL Task-Level Behavior	Notes
End	Quit	None.
Include	MissionScriptInline	Must translate inline file as well.
Set Position	SetPosition	Compute latitude and longitude from Set Position location and offset.
Wait Depth	MetaCommand	Set name attribute to "waitDepth." Set content attribute value to depth.
Wait Magnet	MetaCommand	Set name attribute to "waitMagnet."
Wait Prop	MetaCommand	Set name attribute to "waitProp." Set content attribute to required revolutions per minute.
Wait Run	MetaCommand	Set name attribute to "waitRun."

Table 5.9. Mappings from REMUS UUV Objectives to Single AVCL Task-Level Behaviors

The task-level behavior sequences generated by the REMUS-to-AVCL translator are similar to those generated by the Seahorse-to-AVCL translator in that there are frequently behaviors that are generated only under certain circumstances. In the case of scripts translated from REMUS missions, however, all of the optional behaviors are MetaCommand behaviors that are used to capture information that cannot be represented in AVCL (e.g., the triangle depth and altitude control information). There are no optional

actions in REMUS objectives along the lines of those implied by the Seahorse Surface Comms order Return to Depth and Return to Starting Point parameters.

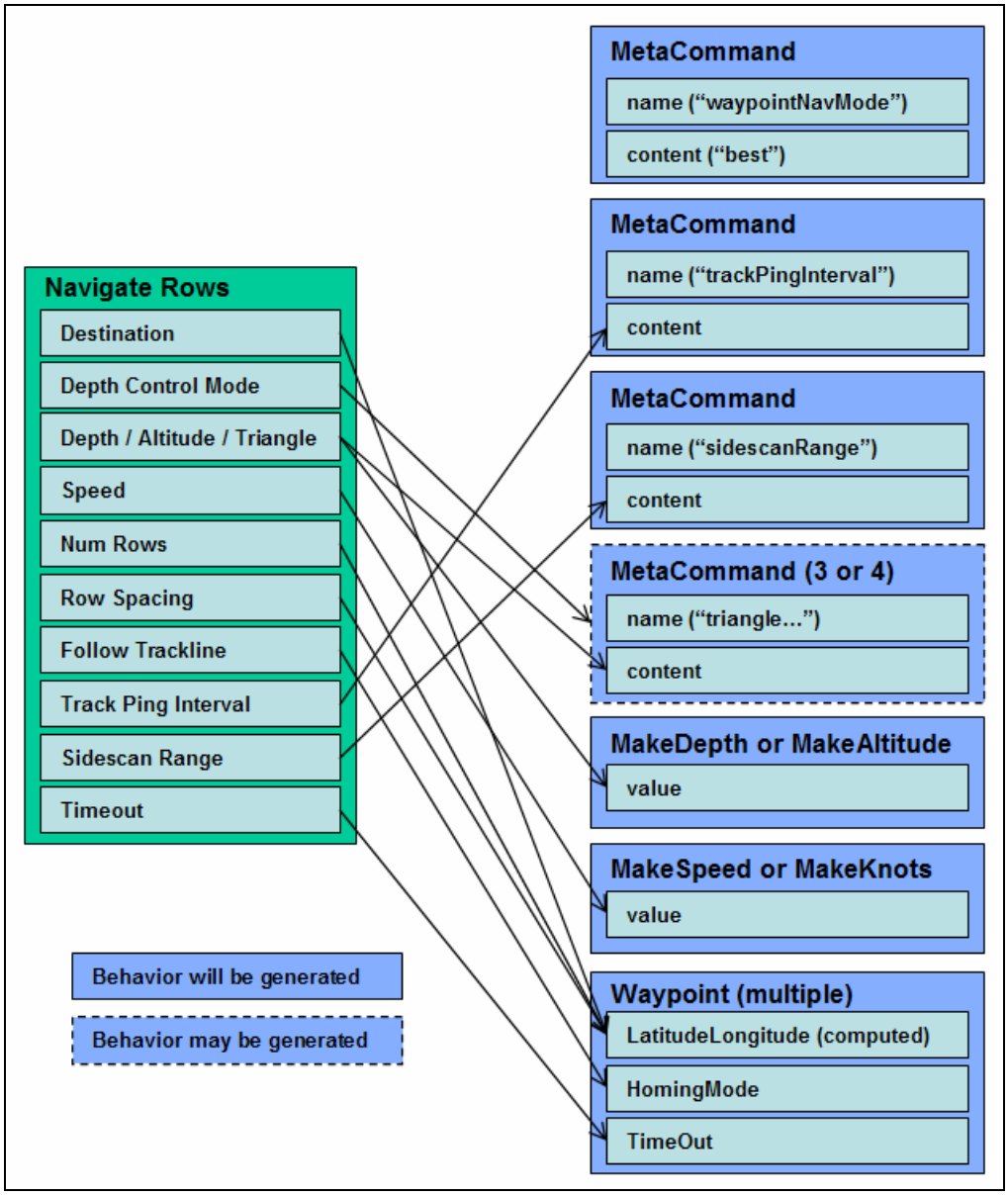


Figure 5.23. Data Mapping from a REMUS UUV Navigate, Dead Reckon or Transponder Home Objective to AVCL Task-Level Behaviors

The AVCL task-level behavior sequences corresponding to REMUS objectives follow the same general pattern as those generated from Seahorse orders. MetaCommand behaviors are generated first, followed by depth and speed related behaviors and finally, waypoint or other behaviors. When possible MetaCommand behaviors intended to influence the semantics of another behavior, such as

MetaCommand behaviors that specify triangle altitude or depth control parameters, are placed immediately prior to the behavior to which they relate. Organization of task-level behavior sequences in this manner accurately captures the semantics of the original REMUS objective and provides for an intuitive series of commands from the perspective of the REMUS objective. It also arranges behaviors so that XSLT stylesheets translating the sequence to vehicle-specific formats can process MetaCommand, depth, and speed related behaviors before waypoint or other behaviors whose requirements they may partially define.

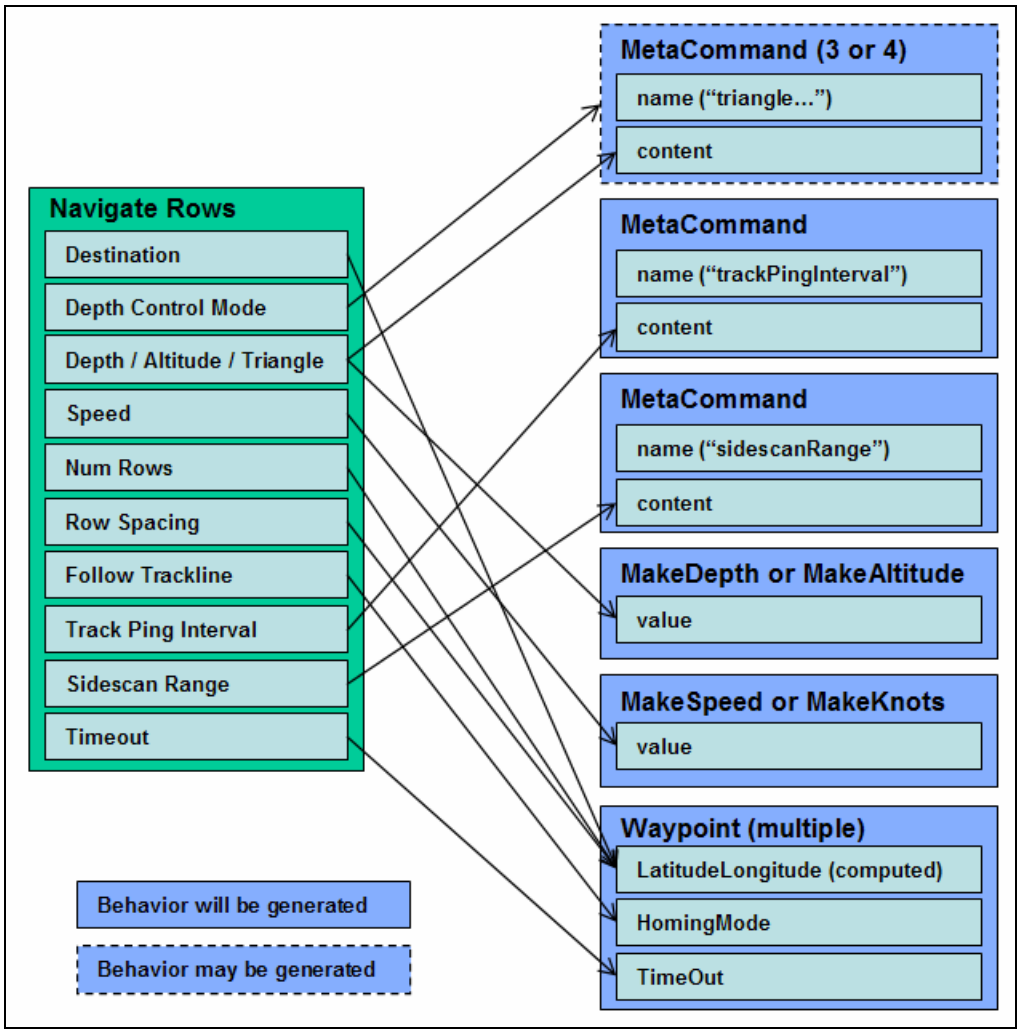


Figure 5.24. Data Mapping from a REMUS UUV Navigate Rows Objective to AVCL Task-Level Behaviors

D. BINARY DATA FORMATS

1. Overview

XSLT and context-free grammars provide straightforward mechanisms for automated translations between common data-model-compliant XML and arbitrary text-based, vehicle-specific data formats. The methods discussed thus far, however, are not directly applicable to binary data formats such as JAUS. In order for the proposed common autonomous vehicle data model to be applied to arbitrary vehicles, a mechanism must be developed for the conversion of AVCL to binary data formats and vice versa. XML is not a binary format, and there are no standards or tools along the lines of XSLT capable of converting XML to an arbitrary binary format or creating XML from binary data. Nevertheless, XML can be used as the basis of a mechanism for both conversions required to support AVCL compatibility with these diverse data formats.

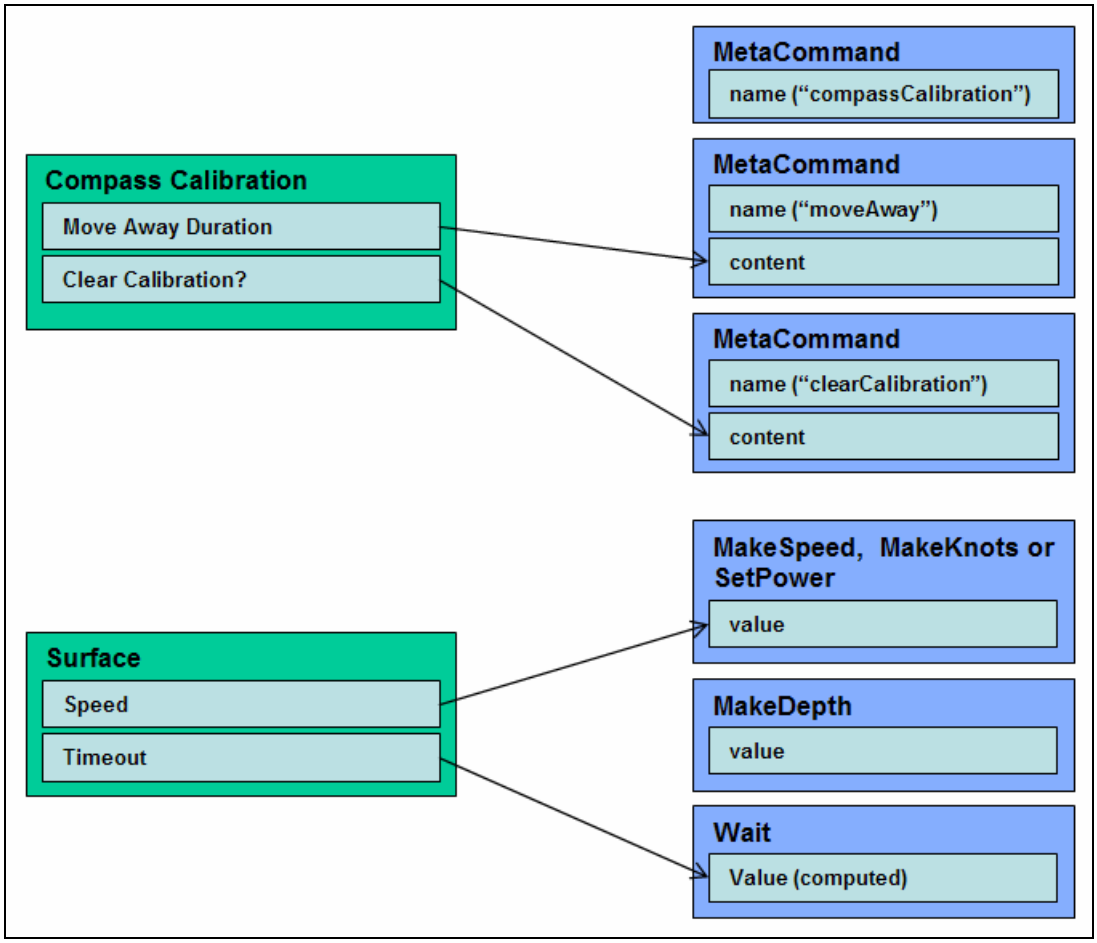


Figure 5.25. Data Mapping from REMUS UUV Compass Calibration and Surface Objectives to AVCL Task-Level

The use of XML with binary data leverages the fact that both XML and binary formats are nothing more than data encoding methods (a principle that is also true of programming objects and context-free languages). Data can be represented in any of these formats and the selection of the appropriate encoding depends on the usage requirements. Binary data is often preferable for communications whereas XML is often used when self-validation, web compatibility or platform independence is desired, and programming objects are used within applications. Methods are also commonly available for converting between various formats. XML data binding and DOM, for instance, convert between XML and programming objects, serializers convert programming objects to binary format for storage or transmission, and readers convert binary data to programming objects.

The proposed method of fostering compatibility between binary formats and AVCL leverages the general equivalence of various data encodings using an XML encoding of the binary data. First the XML form of the binary data is transformed to or from AVCL using XSLT. As indicated in Figure 5.26, which graphically depicts the relationships between AVCL and XML-encoded JAUS messages, an XML schema is defined for the XML version of the binary data format to be encoded and XML data binding is used to create an API corresponding to the schema. A custom reader and serializer can then read and write binary data to and from the schema-based programming objects and XSLT stylesheets are used to convert the schema-governed JAUS-XML documents to and from AVCL. Using this methodology conversion between AVCL and any potentially compatible binary data format requires three steps:

1. Definition of a schema to constrain the XML encoding of the binary data.
2. Development of a reader and serializer to read and write binary data to and from JAXB-derived programming objects.
3. Development of XSLT stylesheets to translate XML documents corresponding to binary data to and from AVCL.

2. JAUS-XML Overview

Unlike AVCL, JAUS does not directly support scripting. Rather, the messages that make up the implicit JAUS command set discussed in Chapter IV are transmitted individually to immediately elicit vehicle activities. There is no JAUS construct along the lines of an AVCL task-level behavior script for grouping a set of command messages

together. This difference between AVCL and JAUS is handled by the reader and serializer, however, and does not pose a problem during translation.

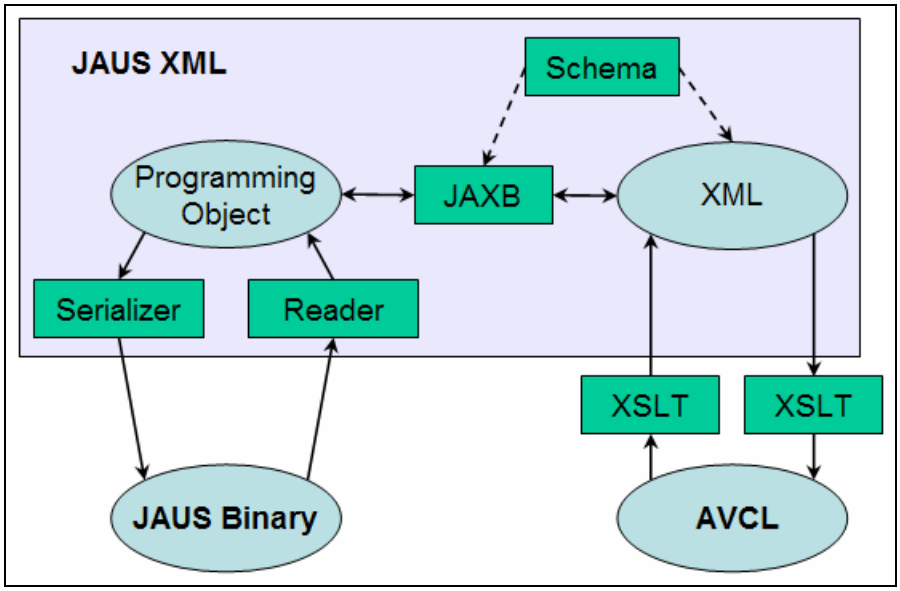


Figure 5.26. XML-Based Translation between JAUS and AVCL

An individual JAUS message is encoded in XML with a Header element followed by a message-type-specific element containing the actual message data. The Header element shown in Figure 5.27 contains all of the information from the JAUS message header fields (with the exception of the command code) and is common to all XML-encoded message types. Data value ranges in the message header element are constrained in accordance with (JAUS, 04-4). Additionally, a number of header fields have fixed values (e.g., the version attribute of the MessageProperties element always has a value of "3.2") and are included for clarity. The command content of the message is encoded as a message-type-specific message data element corresponding to one of the 22 message-types included in the schema. Some of these, such as the QueryHeartbeatPulse element, are empty and serve only to identify the type of message (the command code field of the binary JAUS message header). The message data elements for Query Class messages, contain Boolean attributes identifying the specific data requested by the message. Most message data elements corresponding to Command and Inform Class messages contain child elements with command parameters (Command Class messages) or vehicle state information (Inform Class messages). The content models of message

data elements for corresponding Command and Inform messages (e.g., Set Global Vector and Report Global Vector) are, more often than not, identical.

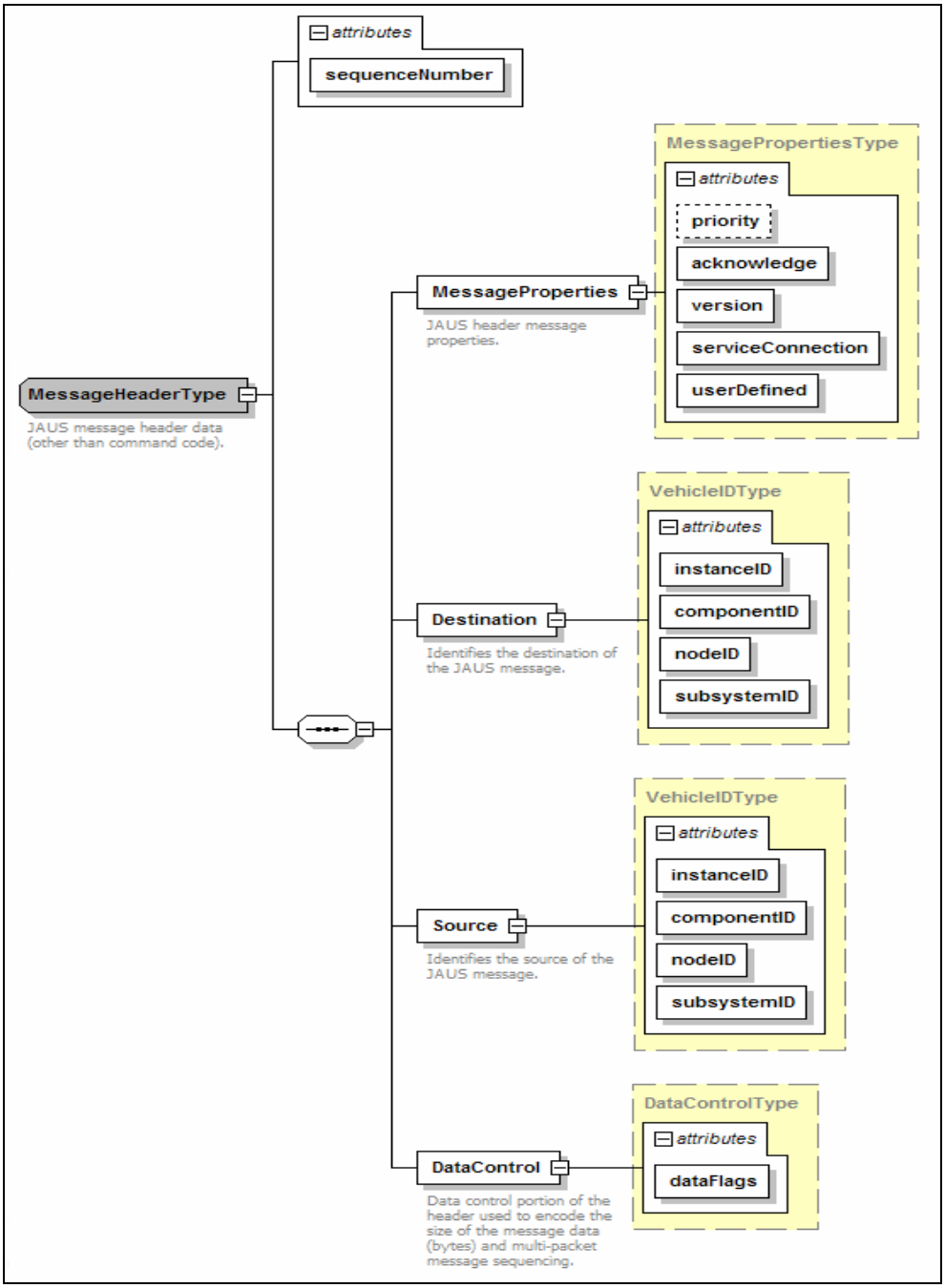


Figure 5.27. An XML Encoding of a JAUS Message Header

The top level element of a JAUS-XML document provides a wrapper for a series of messages. The reader constructs the list according to the content of the binary data

source from which it is reading (input stream, file, etc.) and all readable messages are encoded into a single JAUS-XML document. Similarly, the serializer writes out the contents of the message list in a single serialized stream regardless of whether the list consists of one or many messages. In this way, the list structure of the JAUS-XML document can be used to encode a single JAUS message or a series of JAUS messages.

3. Conversion of JAUS-XML to AVCL

As mentioned previously, an XSLT stylesheet is used to convert XML-encoded JAUS messages to AVCL. In this regard there is little difference between this transformation and the conversion of AVCL task-level behavior scripts to text-based data formats beyond the observation that the product of the JAUS-to-AVCL translation is XML vice free text. There are, however, two potential interpretations of a series of JAUS commands. The most obvious interpretation is to treat the JAUS message sequence literally (i.e., as a series of messages that are to be transmitted or received exactly as they are encoded). On the other hand, JAUS messages are the command mechanism of a JAUS system, so it is reasonable for a vehicle receiving a JAUS message to interpret it as an imperative instruction. Depending on the vehicle involved, it may be more appropriate to remove the message-specific constructs during translation and simply treat a message sequence as a series of commands (i.e., as a task-level script). Since both of these scenarios are feasible, stylesheets supporting both approaches were developed.

As indicated by Figure 5.28, the most notable differences between the two JAUS-to-AVCL stylesheets are the root tag of the generated document (AVCLMessageList or AVCL) and the mapping of the JAUS-XML header. When generating an AVCL message list, JAUS-XML header information is used to generate the AVCL message header, whereas it is included in an AVCL task-level behavior script using MetaCommand behaviors. Both stylesheets map the message data section of Command Class JAUS-XML messages to AVCL task-level behaviors. However, the two mappings each treat the generated task-level behavior differently. When mapped to an AVCL message list, the generated task-level behaviors comprise the body of a message in the list. When mapped to a task-level behavior script they are inserted directly into the script. Query and Inform Class messages are translated to equivalent AVCL messages in the AVCL message list, but they do not map to AVCL task-level behaviors so they are

effectively ignored by the script-generating stylesheet (a warning message is generated and a MetaCommand is inserted into the script).

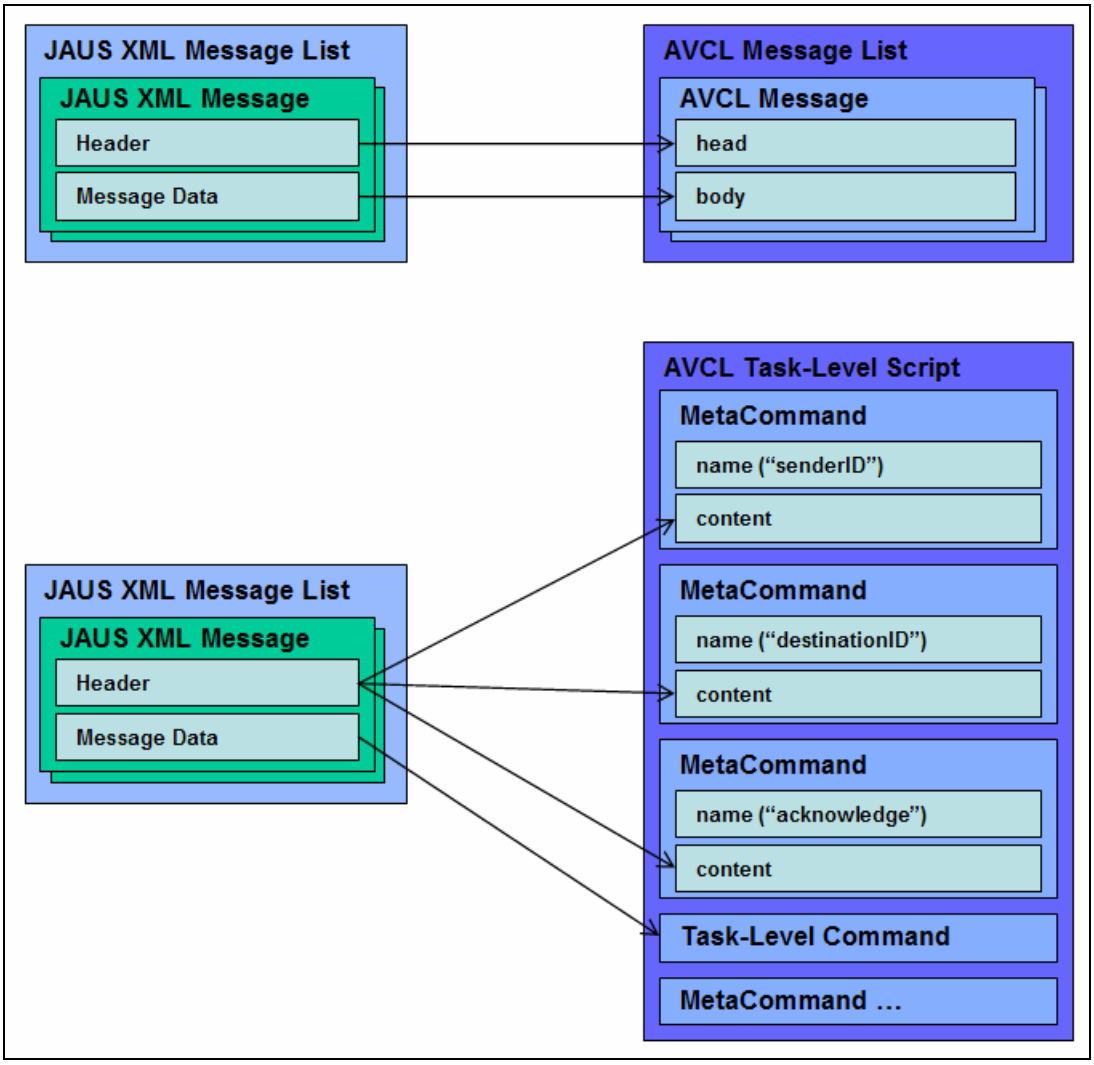


Figure 5.28. Mapping of a JAUS-XML Message List to an AVCL Message List or Task-Level Behavior Script

Of the 22 JAUS messages available in the JAUS-XML schema, 20 can be atomically represented using a single AVCL construct. These are listed in Table 4.11 along with the AVCL construct to which they are mapped. In the case of Command Class messages, the corresponding AVCL construct is a task-level behavior (possibly embedded into the body of an AVCL message). When an AVCL message list is the desired transformation product, Query Class messages are transformed into AVCL InformationRequest messages. Inform Class messages to be included in an AVCL

message list, on the other hand, are transformed into VehicleState, VehicleCharacteristics or empty messages depending on the type of information being transmitted. If the JAUS message sequence is being used as the basis of an AVCL task-level behavior script, the presence of a Query or Inform Class message is indicated by a MetaCommand behavior, but the content of the JAUS message is not incorporated into the task-level script.

Message Type	AVCL Construct	Notes
Query Heartbeat Pulse	InformationRequest message	Information type attribute set to "ping."
Query Global Pose	InformationRequest message	Information type attribute set to "posture."
Query Global Vector	InformationRequest message	Information type attribute set to "posture."
Query Platform Specifications	InformationRequest message	Information type attribute set to "vehicleCharacteristics."
Query Time	InformationRequest message	Information type attribute set to "vehicleTime."
Query Travel Speed	InformationRequest message	Information type attribute set to "velocity."
Query Velocity State	InformationRequest Message	Information type attribute set to "velocity."
Query Wrench Effort	InformationRequest message	Information type attribute set to "controlSettings."
Report Heartbeat Pulse	Empty message	None
Report Global Pose	VehicleState message	VehicleState element contains GeographaphicPosition, VerticalPosition and Orientation elements.
Report Global Vector	VehicleState message	VehicleState element contains vehicle telemetry element.
Report Platform Specifications	VehicleCharacteristics message	None
Report Time	VehicleState message	VehicleState element contains a timestamp only.
Report Travel Speed	VehicleState message	VehicleState element contains a BodyCoordinateVelocity element.
Report Velocity State	VehicleState message	VehicleState element contains a BodyCoordinateVelocity element.
Report Wrench Effort	VehicleState message	VehicleState element contains a vehicle control setting element.
Set Global Waypoint	Waypoint behavior	None
Set Time	SetTime behavior	None
Set Travel Speed	MakeSpeed behavior	None
Shutdown	Quit behavior	None

Table 5.10. JAUS Message Types that can be Mapped to a Single AVCL Construct

The remaining JAUS message types (Set Global Vector and Set Wrench Effort messages) are transformed into AVCL task-level behaviors as depicted in Figures 5.29 and 5.30. The behavior sequence generated for these message types depends on the message content and the target vehicle type (specified as a parameter to the XSLT stylesheet). The elevation element of the Set Global Vector message, for instance, can be used to generate a UAV MakeAltitudeMSL behavior or a UUV MakeDepth behavior. If the generated script is intended for a UGV or USV, the elevation element is ignored.

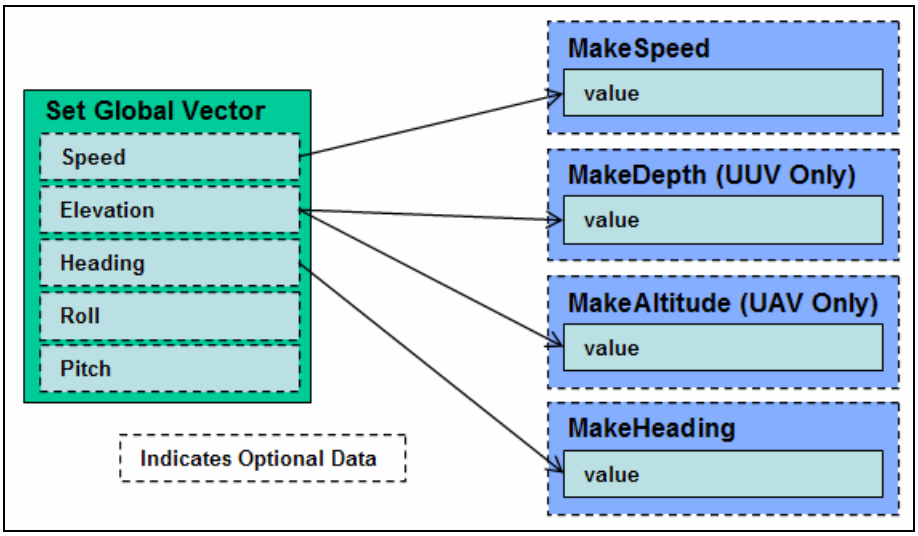


Figure 5.29. Data Mapping from a JAUS Set Global Vector Message to AVCL Task-Level Behaviors

As with other mappings from vehicle-specific data formats to AVCL, JAUS messages often contain a number of optional values. Indicated by a dashed border in Figures 5.29 and 5.30 (a number of examples are also present in the JAUS message types of Table 5.10), the presence of these fields are a prerequisite to the generation of their corresponding AVCL constructs. Also present in a number of JAUS messages are data fields whose values are not carried over to the AVCL constructs during translation (e.g., the Roll and Pitch fields of the JAUS Set Global Vector message of Figure 5.29). MetaCommand behaviors can be used to indicate the presence and values of these data fields if desired, however the common use of these data fields when using JAUS with the envisioned vehicles seems unlikely. For example, the ability of any autonomous vehicle, regardless of type, to maintain a commanded roll or pitch is doubtful, and only UGVs possess the ability to apply the resistive effort (braking) potentially ordered by a Set

Wrench Effort message to any great degree. For this reason most of these data fields are ignored in the JAUS-to-AVCL stylesheets developed for this research.

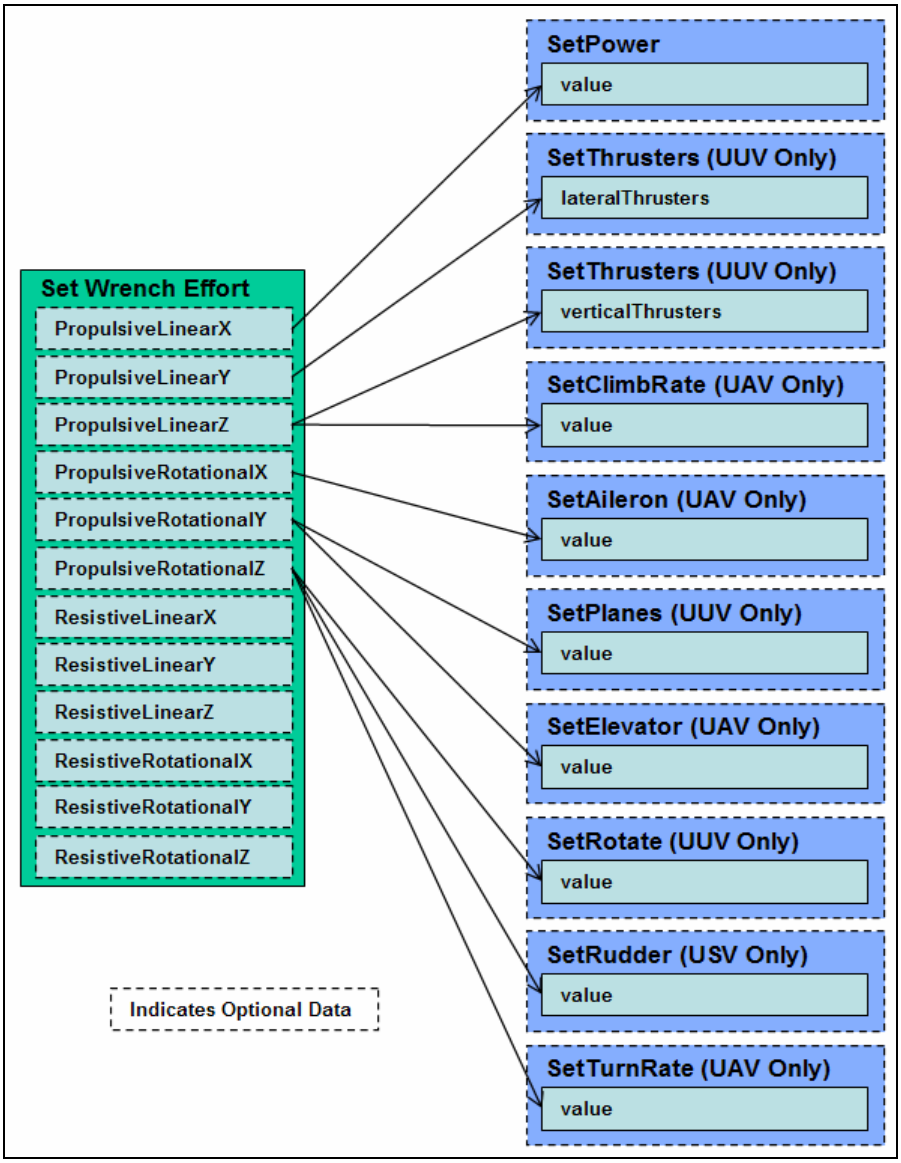


Figure 5.30. Data Mapping from a JAUS Set Wrench Effort Message to AVCL Task-Level Behaviors

4. Conversion of AVCL to JAUS-XML to AVCL

Translation of AVCL task-level behavior scripts, message lists and individual messages to JAUS-XML is also accomplished with an XSLT stylesheet. Regardless of the AVCL document being translated, the result of stylesheet application is a sequence of JAUS-XML messages. Translation of individual AVCL messages and message lists is

slightly less complicated than translation of task-level behavior scripts because the individual messages contain sufficient information to generate the JAUS message header. Translation of task-level behavior scripts, on the other hand, requires specification of the message source and destination, acknowledgement requirements, and priority using parameters to the stylesheet or MetaCommand behaviors embedded within the script. Beyond this, translation of all valid AVCL constructs to JAUS-XML is identical.

AVCL information-request and information-reporting messages are converted to JAUS messages as indicated in Tables 5.11 and 5.12 with each AVCL message being translated to a single JAUS message. Not surprisingly, AVCL messages that are used to request information correlate to JAUS Query Class messages, and messages that are used to report information correlate to Inform Class messages. Translation of AVCL task-level behaviors (and messages with task-level behavior content) is only slightly more involved. As shown in Table 5.13 most task-level behaviors are mapped to a single JAUS message and even the Loiter and Waypoint behaviors are frequently translated to a single message. A Set Travel Speed or Set Wrench Effort message is placed prior to the Set Global Waypoint message if a behavior specifies the transit speed or power respectively.

Information Request Content	JAUS Message
Ping	Query Heartbeat Pulse
Vehicle Characteristics	Query Platform Specifications
Vehicle Time	Query Time
Control Settings	Query Wrench Effort
Posture	Query Global Pose
Velocity	Query Velocity State

Table 5.11. Translation of AVCL Information Request Messages to JAUS Messages

Information Report Content	JAUS Message
Vehicle Characteristics	Report Platform Specifications
Vehicle State (body velocity)	Report Velocity State
Vehicle State (control order)	Report Wrench Effort
Vehicle State (posture)	Report Global Pose
Vehicle State (time stamp only)	Report Time

Table 5.12. Translation of AVCL Information Reporting Messages to JAUS Messages

Examination of Table 5.13 reveals a number of imperfect mappings. This is not unique to this particular translation and is conceptually not dissimilar to AVCL behaviors that do not translate at all to a particular vehicle-specific language (e.g., the AVCL TakeStation behavior does not correlate to a JAUS message). The fact of the matter is that regardless of target vehicle there may be AVCL behaviors that are beyond the vehicle's capabilities. There are typically other task-level behaviors that are within the target vehicle's physical capabilities but whose semantics cannot be fully captured by a vehicle-specific data format. Hover and Loiter behaviors fall into this category when being translated to JAUS messages because they require specific vehicle action upon waypoint arrival, but the JAUS Reference Architecture does not specify what a vehicle is to do upon reaching a waypoint (JAUS, 04-4).

Task-Level Behavior	JAUS Messages	Notes
Hover	Set Global Waypoint	UUV only. Hover behavior upon arrival not guaranteed.
Loiter	Set Travel Speed (optional) Set Wrench Effort (optional) Set Global Waypoint	Loiter behavior upon arrival not guaranteed. Set travel speed used if transit speed specified. Set wrench effort (propulsive linear X field) used if transit speed specified.
MakeAltitudeMSL	Set Global Vector	UAV only. Sets the elevation field.
MakeClimbRate	Set Wrench Effort	UAV only. Sets the propulsive linear Z field.
MakeDepth	Set Global Vector	UUV only. Sets the elevation field.
MakeHeading	Set Global Vector	Sets the heading field.
MakeKnots	Set Travel Speed	None
MakeSpeed	Set Travel Speed	None
MakeTurnRate	Set Wrench Effort	UAV only. Sets propulsive rotational Z field.
MoveLateral	Set Wrench Effort	UUV only. Sets propulsive linear Y field
MoveRotate	Set Wrench Effort	UUV only. Sets propulsive rotational Z field.
Quit	Shutdown	None
SetAileron	Set Wrench Effort	UAV only. Sets propulsive rotational X field.
SetElevator	Set Wrench Effort	UAV only. Sets propulsive rotational Y field.
SetPlanes	Set Wrench Effort	UUV only. Sets propulsive rotational Y field.
SetPower	Set Wrench Effort	Sets the propulsive linear X field.
SetRudder	Set Wrench Effort	UUV, USV, and UAV only. Sets propulsive rotational Z field.
SetThruster	Set Wrench Effort	UUV only. May set propulsive linear Y field OR the propulsive linear Z field.
Waypoint	Set Travel Speed (optional) Set Wrench Effort (optional) Set Global Waypoint	Set travel speed used if transit power specified. Set wrench effort (propulsive linear X field) used if transit speed specified.

Table 5.13. Translation of AVCL Task-Level Behaviors to JAUS Messages

As discussed earlier in this chapter, the stylesheet used for translation of AVCL to any vehicle-specific format can handle task-level behaviors that do not map perfectly to vehicle-specific commands in one of two ways. If clearly incompatible with the vehicle-specific format, the stylesheet needs to generate error messages or initiate recovery procedures. Alternatively, the task-level behavior can be converted to the best vehicle-specific fit as was done when converting the AVCL Hover and Loiter to JAUS messages. The inability to ideally map all AVCL behaviors to a particular vehicle-specific format does not indicate an inherent incompatibility, but must be dealt with during translation nevertheless. The requirements of the particular situation will dictate whether a mapping from a particular behavior to the target data format is reasonable or not.

E. SUMMARY

Two important requirements for the use of a common autonomous vehicle data model to support arbitrary vehicles is the ability to translate data-model-compliant data to formats suitable for specific vehicles, and conversely to convert vehicle-specific data to a data-model-compliant form. Further, the processes involved in these translations must be general enough to apply to arbitrary vehicle formats. This capability is demonstrated through the automated translation of AVCL to and from four text-based vehicle-specific data formats and one binary format.

Generation of arbitrary text formats from AVCL is accomplished using XSLT stylesheets. Semantic and syntactic differences between AVCL and various vehicle-specific formats are handled by implementing a novel design pattern to obtain the functionality of mutable variables within XSLT stylesheets, accessing external functions through the use of XSLT extensions to implement functionality not normally available in XSLT, and the inclusion of MetaCommand behaviors in AVCL task-level behavior scripts to capture information that is truly vehicle-specific.

The more difficult problem of translating text-based data formats from arbitrary vehicles to AVCL is accomplished through the use of context-free grammars, the Cocke-Younger-Kasami parsing algorithm, and depth-first traversal and translation of a context-free-grammar-based parse tree. Analogous to the use of XML Schema to define the content and structure of an XML document and XSLT to transform documents to other text-based formats, a context-free grammar constrains the content and structure of valid

non-XML text documents, while the Cocke-Younger-Kasami algorithm and depth-first translation transforms compliant native-vehicle documents to another text-based format,.

Translation of binary data formats along the lines of JAUS messages are translated to and from AVCL using an intermediate XML encoding of the binary data. A binary data instance is converted to AVCL by implementing a data-format-specific binary reader that loads binary instances into data-bound programming objects conforming with the XML encoding. The data-bound object is marshaled through an XSLT stylesheet designed to convert the XML encoding to AVCL. Similarly, an XSLT stylesheet can be applied to an AVCL document to transform it into an XML encoding of the target binary format. The actual binary form is generated by unmarshalling the XML document into a data-bound programming object that is serialized to the desired binary format by a writer implemented for the purpose.

The successful implementation of the translations discussed in this chapter demonstrates how an XML-based common autonomous vehicle data format can be used as a bridge between arbitrary vehicle-specific formats. Whether text-based or binary, instances of any vehicle-specific format can be converted to common-data-model-compliant XML. Similarly, common-data-model-compliant data can be converted to arbitrary text-based or binary vehicle-specific formats. In effect this means that data in any vehicle-specific format can be converted to any other vehicle-specific format by using the common data model as an intermediate form—one of the primary objectives of this research.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. OFF-VEHICLE DECLARATIVE MISSION APPLICATION

A. INTRODUCTION

One aspect of the translation forms discussed in Chapter V is that they apparently deal solely with the task-level behavior and messaging portions of the AVCL schema. In theory the translations discussed in Chapter V can be applied to declarative missions as well, but the fact of the matter is that the current generation of autonomous vehicles rely almost exclusively on a level of control that maps more naturally to the AVCL task-level behavior set. This state of affairs might call into question the usefulness of a more abstract declarative mechanism such as the one developed in this work for defining autonomous vehicle tasking.

This chapter begins to address the issue of declarative mission use by demonstrating how declarative autonomous vehicle mission specification can be used in conjunction with task-level behavior scripts during the pre-mission phase of an operation. Specific topics include the generation of task-level behavior scripts from declarative missions and the inference of appropriate declarative agendas from unannotated task-level behavior scripts.

B. GENERATION OF TASK-LEVEL BEHAVIOR SCRIPTS FROM DECLARATIVE SPECIFICATIONS

1. Overview

The first application of the declarative missions is their pre-mission use as the basis for task-level behavior scripts that are subsequently to be used for vehicle tasking. In many instances, it might be preferable to generate a task-level behavior script from a declarative AVCL agenda rather than to develop it manually because of the relative simplicity of the declarative tasking, since it is often more intuitive to describe what a mission is intended to accomplish than to define the precise detailed steps describing how to proceed. It is more straightforward, for instance, to simply direct a vehicle to search an area than to define a long series of waypoints that provide the appropriate coverage. A number of mechanisms are potentially applicable, including traditional artificial intelligence search and planning algorithms as well as heuristic application of predefined or parametrically-derived behavior sequences. Before covering the actual mechanisms

used to convert declarative agendas to task-level behavior scripts, however, a number of issues particular to the pre-mission script generation bear discussion.

Most obviously, the success or failure of the individual goals of a declarative mission is not known in advance, so the state transitions that are required over the course of the mission are not known. In short, although it is possible to convert individual goals into suitable task-level behavior scripts, it is not possible to definitively determine how to appropriately order the scripts. If, however, the assumption is made that all goals will succeed, a script representing a best-case mission progression can be generated. The script-generation methodologies discussed in the remainder of this section are based on this assumption. However, this assumption applies only to pre-mission script generation and not to the in-mission planning and replanning discussed in Chapter VII.

A closely related issue concerning pre-mission conversion of declarative agendas to task-level behavior scripts (and groups of sub-scripts) arises from the static nature of scripts. As with the sequencing of sub-scripts corresponding to individual goals in an agenda finite state machine, the makeup of the sub-scripts themselves is fixed once generated. While this is typically true of the scripts (i.e., they do not change once the vehicle begins the mission), it does make effective pre-mission use of some goal-types more difficult. AVCL goal types such as Attack, Demolish and MarkTarget, for instance, require the vehicle to locate a target before performing the specified action, but the precise position in the script at which the target will be located is not known when the script is generated in advance. A more subtle aspect of the static nature of scripts is that it may not be possible to determine when a goal has completed. An area search for a single target, for instance, is theoretically successful when the target is located. The search script, however, provides for coverage of the entire area even if the target is promptly encountered on the first leg of the search. It must be emphasized, however, that this problem is not unique to scripts generated from declarative agendas since it will also prove to be the case for manually generated area search scripts. As with goal ordering, these potential behavior script ordering issues apply only to pre-mission script generation.

A more insidious problem inherent in the use of AVCL agendas is that some information contained in the goals cannot be expressed accurately using task-level

behaviors. In fact, some goal types implicitly require activities that cannot be commanded using the existing task-level behavior set (e.g., there is no task-level behavior available to command the vehicle to decontaminate an area). This is particularly true of goal types that require situational use of mission-specific systems. Some current and developmental autonomous vehicles provide for limited control of mission systems, but the diversity of potential onboard systems and the lack of a mission payload standard make mission-system control using a generalized command set problematic. From a control standpoint, many command languages implicitly assume that the vehicle "knows" its purpose and can manage its mission systems accordingly. For instance, a REMUS UUV with an onboard Computer-Aided Detection / Computer-Aided Classification system will always look for and classify submerged mine-like objects, so there is no need to incorporate this sensor tasking into the mission definition (although REMUS objectives do provide some control over the sidescan sonar settings).

A number of current research efforts may provide the promise of standardized mission system interface. Examples include JAUS (the Manipulator and Environment Sensor subgroups of the Command, Query and Inform Message Classes), the Open Geospatial Consortium's Sensor Model Language (OGC, 06) and ASTM International's Standard Guide for Unmanned Undersea Vehicle Mission Payload Interface (ASTM, 06). The further development of these standards will likely make it possible to more effectively incorporate mission-system management into the autonomous vehicle control architecture, thereby making more robust execution of many goal-types possible. Until then, the functionality required to fully execute many missions defined as declarative agendas in different robot architectures will remain elusive.

In the near term, there is no good workaround that provides for the control of mission-specific systems using the vehicle's command script—even if incorporated into the AVCL task-level behavior set, mission-specific behaviors are unlikely to translate to vehicle-specific commands. Similar to their use in the translations of Chapter V, MetaCommand behaviors are used by the planners discussed in this and the next chapter to indicate mission-system control requirements. To the extent that the command language of the target vehicle provides for mission-system control, these MetaCommand behaviors can be converted to vehicle-specific commands. If the mission requirements

are beyond the capabilities of the target vehicle, the presence of the MetaCommand behavior can be used to generate a warning or error. In all other cases, it is simply assumed that vehicle's mission systems are operated correctly by default.

Notwithstanding the aforementioned issues, it is possible to convert most AVCL agendas into task-level behavior scripts closely resembling those that might be manually developed to accomplish the same goals. It is helpful to the pre-mission script-generation process to assume that individual goals always succeed, that sub-scripts run from start to finish uninterrupted, and that mission-specific systems aboard the target vehicle operate in a manner consistent with the goals without being explicitly ordered to do so. Accordingly, the methods discussed in the remainder of this section make use of AVCL MetaCommand behaviors to embed goal-specific information in the script when appropriate. For example, the script generated to accomplish a Search goal includes a waypoint pattern over the area that provides the requisite probability of detection. The entire script executes from start to finish uninterrupted after which the script corresponding to the next goal can begin. Finally, the script includes MetaCommand behaviors describing the objectives of the search, but it is assumed that the autonomous vehicle will use its sensors and mission systems to locate the correct targets regardless.

2. Decision-Tree-Based Generation of Task-Level Behavior Scripts

The process by which autonomous vehicle missions are manually developed is not significantly different than the process by which manned or remotely operated vehicle missions are planned. The process typically begins with a formal or informal analysis of the mission's objectives, the capabilities of the vehicle, and the characteristics of the operating area, then ends with the development of a suitable mission script. Utilizing the same information that a human operator might use in designing a mission, this process can be automated for the goal-types available in AVCL. Complete planning for the accomplishment of a specific goal can be divided into two steps: planning the transit to the operating area and planning the in-area activities to successfully accomplish the goal. The remainder of this section covers goal-specific, in-area planning and requirements for global path-planning between operating areas is discussed later in this chapter.

A straightforward mission-planning methodology that can be applied to the autonomous vehicle domain relies on a set of Boolean propositions describing

characteristics of the operating area and goal. Inference rules implemented as a decision tree are then used to correlate the propositions to a behavior-sequence template appropriate to the circumstances. When the template is applied, vehicle and area characteristics are incorporated to generate a specific sequence of task-level behaviors that can be executed to accomplish the goal.

As an example, consider the AVCL goal of Figure 6.1 calling for the search of a rectangular area. The inference rules corresponding to an area search goal with a rectangular (but not square) operating area are able to determine a suitable search pattern that can provide the requisite coverage, consisting of a series of parallel lines that progress from one side of the operating area to the other as depicted in Figure 6.2. This pattern is formally referred to as a parallel-track pattern in the International Aeronautical and Maritime Search and Rescue Manual (IMO and ICAO, 98). The parallel-track template generates the specific waypoint series based on area length, width and orientation and vehicle sensor sweep width—defined in the Navy Search and Rescue Tactical Information Document as the distance "obtained by reducing the maximum detection distance…so that scattered targets which may be detected beyond the limits are equal in number to those which may be missed within those limits" (CNO, 97).

```
<Goal id="Goal1">
   <Search datumType="area" requiredPD="0.8"/>
   <OperatingArea>
      <Rectangle>
         <NorthwestCorner>
            <XYPosition x="75000" y="-27500"/>
         </NorthwestCorner>
         <Width value="25000"/>
         <Height value="75000"/>
         <Orientation value="30"/>
      </Rectangle>
   </OperatingArea>
   <Timing start="1800" stop="5400"/>
</Goal>
```

Figure 6.1. An AVCL Goal Calling for the Search of a Rectangular Area with a Required Probability of Detection of 0.8

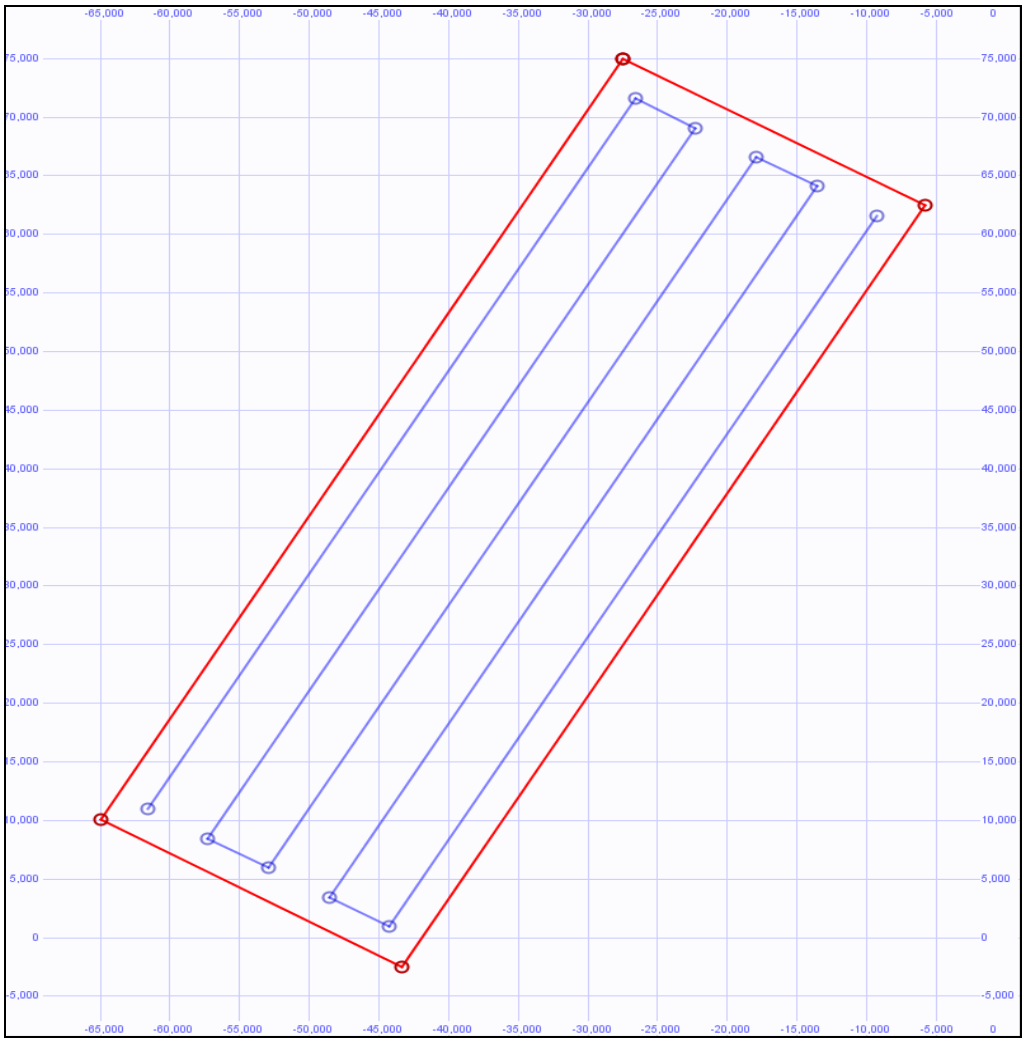


Figure 6.2. A Parallel-Track Search Pattern that can be Executed by a UAV to Accomplish the AVCL Goal Specified in Figure 6.1

For a number of goal types, the term "decision tree" is a bit misleading since only one template (i.e., one general method of accomplishing the goal) was developed during this work. A single template obviously simplifies the task-level behavior sequence generation in that it does not require the use of propositions or inference rules, although vehicle, area and goal characteristics are still used to generate specific task-level behavior sequences. Goals that require the vehicle to jam or monitor electronics transmissions, illuminate an area or relocate to another geographic position all fall into this category. MonitorTransmissions, IlluminateArea and Jam goals, for instance, are accomplished simply by transiting to the center of the designated operating area and turning on the appropriate sensor, illuminator or jammer. MetaCommand behaviors are used to indicate

sensor, illuminator or jammer on and off times. Reposition goals, on the other hand require no goal-specific planning whatsoever since they are considered successful when the vehicle completes the transit to the new operating area. From a pre-mission standpoint, planning for Rendezvous goals is the same as for Reposition goals (with the addition of MetaCommand behaviors identifying the rendezvous target) since the actual location of the rendezvous target within the operating area is not known ahead of time.

Planning for the remaining goal types is somewhat more complicated since there are a number of potential ways that each can be accomplished. However, they do share one significant aspect that greatly simplifies the overall process. Inherent in the accomplishment of each of these goal types is the requirement to search (or methodically cover) the operating area. For instance a patrol goal uses a repeating coverage pattern of the operating area, and a SampleEnvironment goal is accomplished by sweeping the area while the environmental sensors accumulate data. Impracticality of pre-mission script generation notwithstanding, even Attack, Demolish and Decontaminate goals require a search for the target or contaminant. For the purpose of pre-mission script generation, therefore, scripts for the accomplishment of these goal types can be generated by tailoring a search decision tree depicted in Figure 6.3 to match specific goal requirements.

In addition to the parallel-track pattern, the International Aeronautical and Maritime Search and Rescue Manual defines a number of patterns that might be utilized by autonomous vehicles to provide search coverage for a prescribed area. The search patterns described in Figure 6.4, either defined in or adapted from the International Aeronautical and Maritime Search and Rescue Manual, form the basis of the search pattern templates of the search planning decision tree of Figure 6.3.

The first branch of the decision tree is determined by the focus of the search, specifically, whether the search is to focus on a single point (the centroid of the area) or provide for equal coverage of the entire area. The AVCL Search goal explicitly specifies this value using the datumType attribute (which can have a value of either “point” or “area”). Searches corresponding to all other AVCL goal types provide for coverage of the entire area. Propositions used in the remainder of the decision tree are based on the characteristics of the area to be searched and the sweep width of the vehicle sensor.

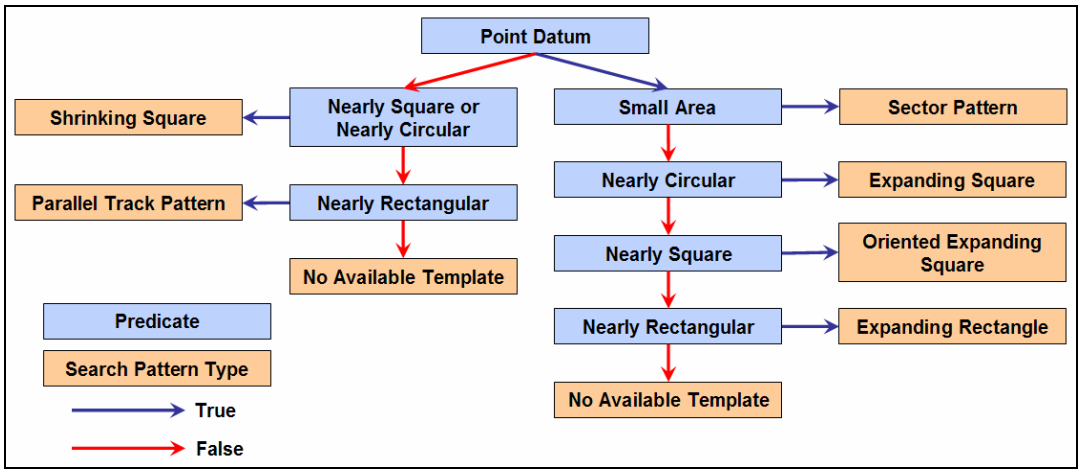


Figure 6.3. A Decision Tree for Determining an Appropriate Search Pattern Based on the Characteristics of the Vehicle, Operating Area, and Search Type

Parallel-track pattern: a sequential series of equally spaced parallel tracks. This pattern is useful in providing for uniform coverage of a large area when the search target can be located anywhere in the area with equal probability.

Expanding-square pattern: a pattern starting at the center of a square area and expanding outward through a series of 90 degree turns and increasingly long legs. This pattern provides uniform coverage of the search area and is useful when the search area is potentially large, but the search target is most likely located near the center of the area.

Sector pattern: a pattern starting at the center of a circular area and dividing the area into pie-shaped wedges using a series of outbound and inbound legs. This pattern provides repeated coverage of the center of the area with less coverage at the edges and is useful for thorough coverage of small areas when the search target is most likely located near the center.

Expanding-rectangle pattern: similar to an expanding-square pattern, this pattern expands at different rates as proscribed by the length to width ratio of the rectangle. This pattern provides thorough (but slightly uneven) coverage of a rectangular area. It is useful in the search of potentially large areas when the search target is most likely located near the center of the area.

Shrinking-square pattern: starts at the outer edge of the area and proceeds towards the center through a series of 90 degree turns and increasingly short legs (essentially the reverse of an expanding-square pattern). This pattern provides uniform coverage of the area and can be used when the search target can be located anywhere in the area with equal probability or if it is most likely located away from the center of the area.

Figure 6.4. Preplanned Search Patterns Available for use in Accomplishing AVCL Goals (After: IMO and ICAO, 98)

For point-focus searches, the next proposition in the tree is dependent on the radius of the area's bounding circle relative to the track spacing (computed using Equation 6.1) of the search pattern. If the bounding circle radius is less than twice the track spacing, then a sector pattern can be used to provide an efficient and highly focused search. For larger areas that are roughly circular (specified as a circle or the search area to bounding circle area ratio is greater than 0.7), an expanding-square pattern with the first leg aligned with the vehicle's inbound heading is used. If the area is roughly square (oriented bounding box length to width ratio greater than 0.8 and area-to-oriented-bounding-box area ratio greater than 0.7) then an oriented-bounding-box-aligned expanding-square pattern is used. Finally, if the area is rectangular, but not square (area to oriented bounding box area ratio greater than 0.7) then an oriented-bounding-box-aligned expanding-rectangle pattern is used. If none of these proposition values is true, none of the predefined patterns applies. The method for generating search patterns for these areas is discussed in the next section.

$$S = 4\left(\frac{W}{P_D} - W\right) \qquad \text{(Eq. 6.1)}$$

The decision process for area searches is significantly simpler. If the area is roughly square or roughly circular, then an oriented-bounding-box-aligned shrinking-square pattern is used. If it is roughly rectangular but not square, a parallel-track pattern is used. If the area is irregularly shaped, then none of the predefined patterns is applicable, so the planning techniques discussed in the next section are used instead.

Once the type of pattern to be utilized has been determined, the actual waypoints can be generated without difficulty. The required probability of detection and the vehicle's sensor sweep width (W) are used to determine the search pattern track spacing (S) using Equation 6.1. Based on the tables in (CNO, 97), Equation 6.1 indicates that a single-search probability of detection of 0.8 calls for a track spacing equal to the sweep width. The track spacing and the oriented-bounding-box of the operating area are used to determine the number of search legs are required by the pattern. The final step in waypoint generation is to clip out-of-area legs to keep the vehicle within the boundaries of the operating area. The result of out-of-area clipping is a pattern along the lines of the

rounded-corner-out expanding-square pattern depicted in Figure 6.5 corresponding to a point-focused search of a circular operating area.

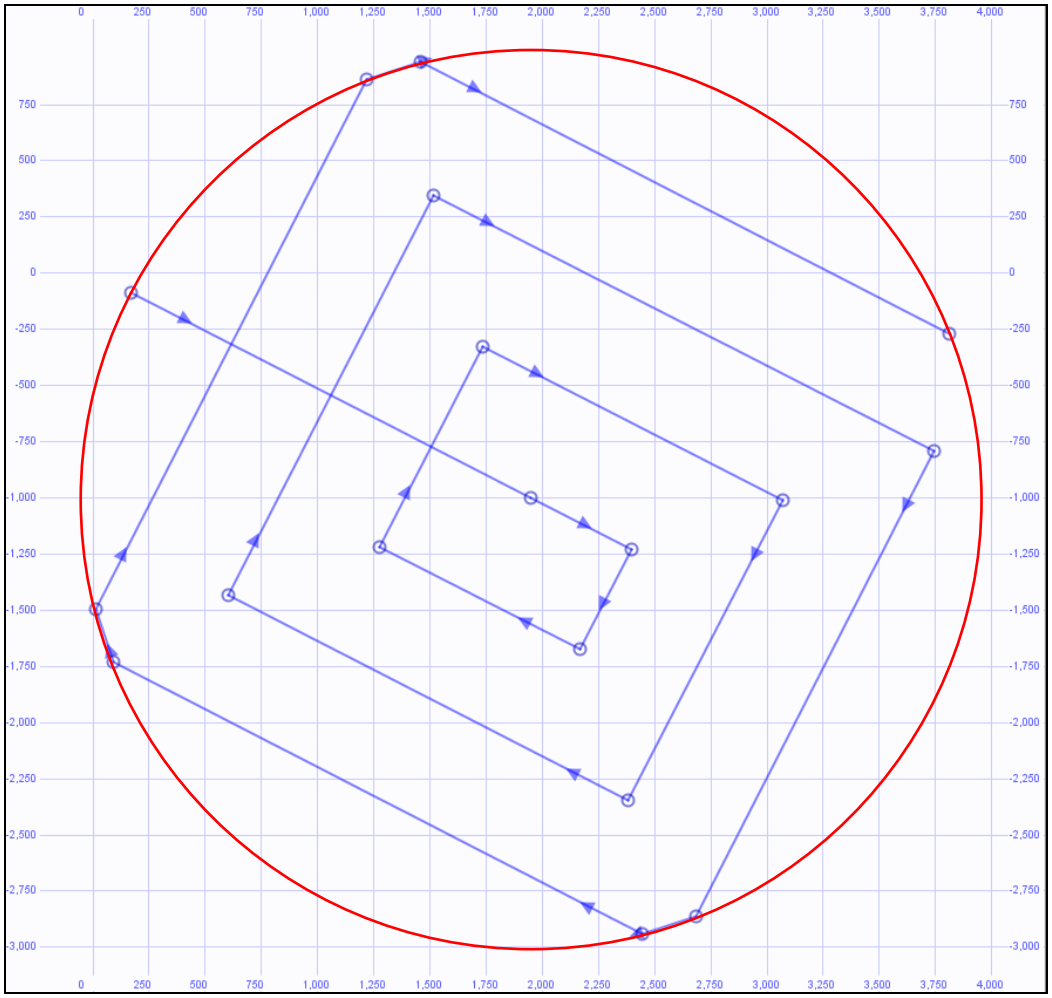


Figure 6.5. An Expanding-Square Search Pattern for use by a USV in Accomplishing a Point-Focused Search of a Circular Operating Area

3. Use of Planner-Generated Search Pattern Scripts

a. Overview

As the decision tree of Figure 6.3 indicates, there are situations to which none of the search patterns discussed in the preceding section can be easily applied. These cases typically involve operating areas consisting of irregularly shaped (most frequently concave) polygons along the lines of the area depicted in Figure 6.6 that are not conducive to the uniform and repeating nature of predefined search patterns.

A number of options are potentially available to address this issue. Among the most obvious is to generate a predefined search pattern to fit the operating area's oriented bounding box and clip the search legs that cross the area boundaries. This approach is, in fact, how patterns are generated for operating areas that are only "slightly" irregular when the decision tree of Figure 6.3 is used. For more irregular areas, however, these patterns can be inefficient since they often repeatedly transit some parts of the area in order to provide coverage for more remote sections. Further, when the operating area is concave, it is possible for search legs to begin and end within the area but go outside the area during transit, increasing the complexity of segment clipping and further decreasing the efficiency of the final search pattern. Both of these phenomena are illustrated in Figure 6.6, particularly in the eastern portion of the area where the edge is transited multiple times to ensure coverage of the northeastern and southeastern corners.

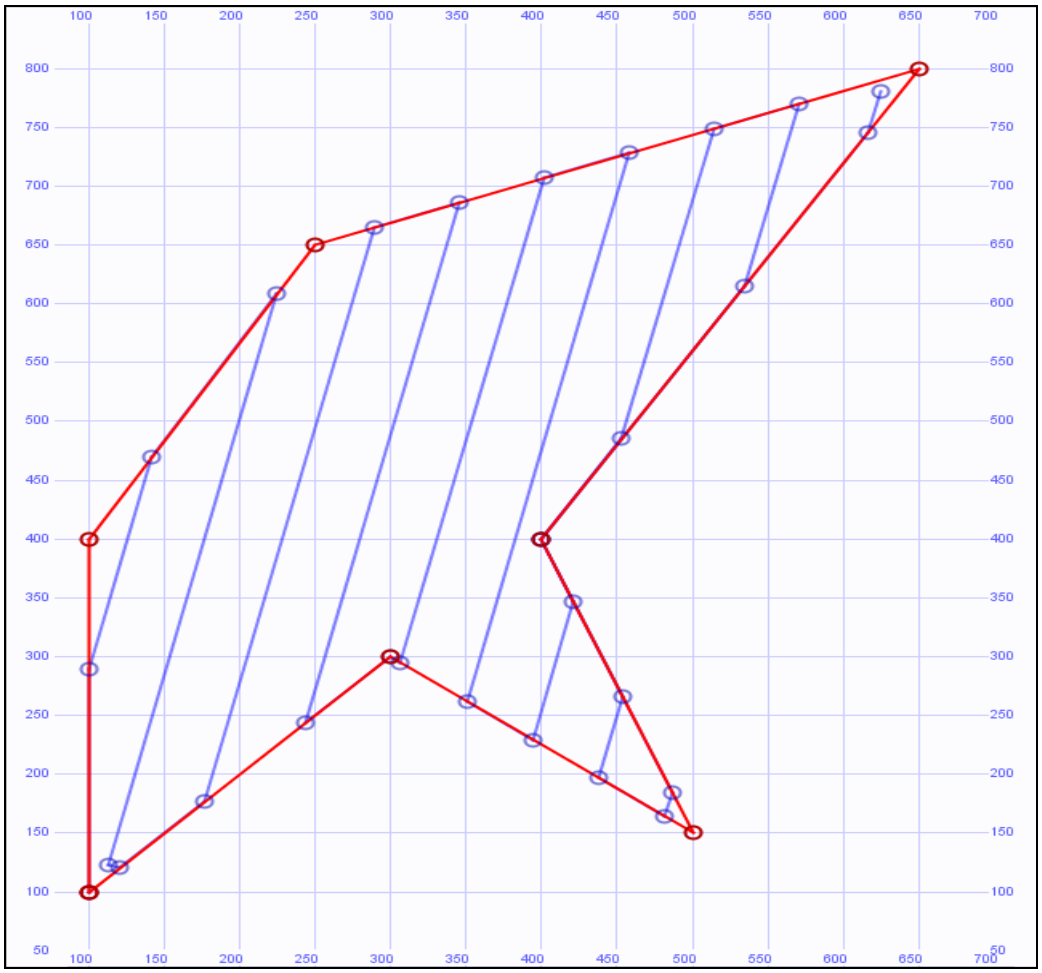


Figure 6.6. An Irregularly Shaped Operating Area and Overlaid Parallel-Track Search Pattern that has been Adjusted to Avoid Out-Of-Area Excursions

Another potential approach might be to implement additional predefined patterns to more naturally fit more complex shapes. Ultimately, however, this amounts to a workaround that might mitigate the difficulty of fitting predefined search patterns to arbitrary polygonal operating areas, but does not eliminate it. Increasing the number of available patterns and increasing the complexity of the decision tree may be enable the application of further predefined patterns to more complex areas, however areas for which no predefined pattern is ideal may still be encountered on occasion.

A more flexible and universally applicable solution is to incorporate a methodology for planning a search pattern specific to the operating area into the decision tree as depicted in Figure 6.7. The planner must quickly generate a sequence of waypoints that provide effective coverage (i.e., commensurate with the required probability of detection) of areas for which preplanned patterns are not well-suited. Numerous artificial intelligence planning and search methods might be applied and upon inspection, the AVCL task-level behavior set appears suitable for this purpose. The waypoint behavior in particular has inherent postconditions (i.e., the vehicle is at a new position and area around the path is searched) that can be used to define a search graph.

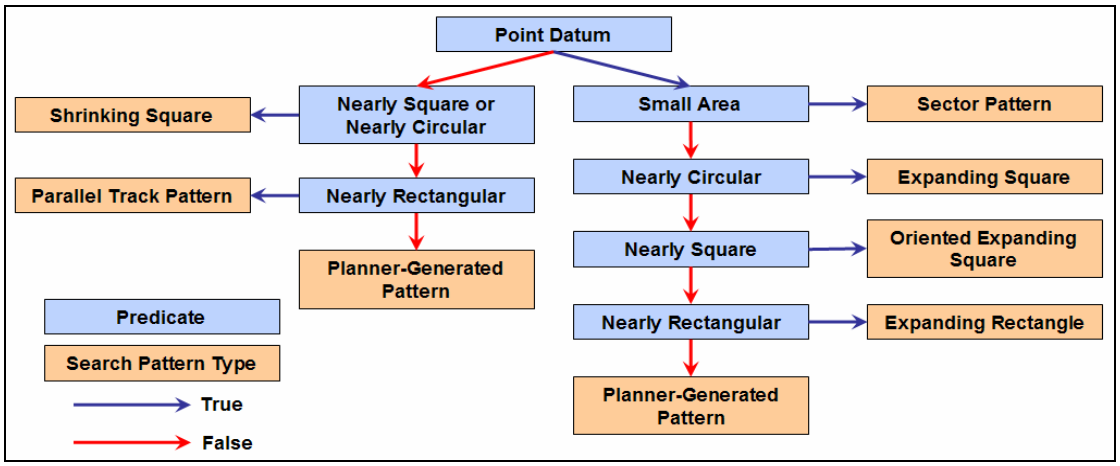


Figure 6.7. A Decision Tree for Determining an Appropriate Area-Search Pattern that Relies on Artificial-Intelligence Planners for Irregularly Shaped Areas

A number of planning and search methods including GraphPlan, breadth-first search, branch-and-bound search, A* and hill-climbing search were considered. Of these, search-graph depth and high branching factors ruled out the uninformed search algorithms (i.e., those that consider all transitions equal and do not order state evaluation

based on their relationships to the goal) such as GraphPlan and breadth-first search. A pure branch-and-bound approach proved impractical for similar reasons, as did many A* implementations. Ultimately, potentially acceptable performance was obtained using two A* search implementations, a hill-climbing search, and a heuristic approximation of the Traveling Salesman Problem. Further analysis follows.

b. A-Star (A) Based Search-Pattern Development*

The underlying structure of all three search-based candidate algorithms is identical. The search area is covered with a hexagonal grid centered at the area's centroid with the diameter of each hexagon set to 0.75 times the computed track spacing (S). A search-graph state is defined by the position of the last waypoint in the candidate search pattern, the heading (ψ) of the vehicle upon reaching the location, and the list of hexagons that have been visited by the partial pattern. A hexagon is considered visited if the pattern's track passes within $S/2$ meters of the hexagon's center. Potential successor states consist of all states that correspond to a vehicle transit from its current position to the center of a hexagon that has not yet been visited. The set of goal states consists of all states with no unvisited hexagons. The search pattern, therefore, consists of the sequence of vehicle positions in the state sequence from the start to the goal.

The basic A* search relies on two metrics: the actual cost of a partial path to a state, and the estimated remaining cost from that state to the goal. The partial path cost is computed simply as the sum of the cost of individual steps. For this A* implementation, the step cost for a transition from state X_i to state X_j is defined by Equation 6.2 where $distance(X_i,X_j)$ is the Euclidean distance between the locations corresponding to X_i and X_j, and $\Delta\psi$ is the magnitude of the turn required at the beginning of the transit. The justification for the distance portion of the cost is fairly obvious, while the turn element of the equation biases the search to favor straighter paths. The estimated remaining cost function is defined by Equation 6.3 where k is a constant and $hexagon(X)$ is the number of hexagons visited as of state X. The constant, k, is used to bias the search in favor of a particular type of solution. Lower values for k favor low-cost solutions at the expense of increased search time (setting k to 0 results in a pure best-first search) while higher values bias the search towards solutions that require fewer steps. The A* implementation tested here uses a k of 5.0 and heavily biases the search towards patterns

that visit the most hexagons with the fewest number of waypoints, resulting in solutions similar to the pattern depicted in Figure 6.8.

$$C(X_i, X_j) = distance(X_i, X_j) + \frac{|\Delta\psi|S}{10} \qquad \text{(Eq. 6.2)}$$

$$E(X_i) = kS(hexagon(X_{Goal}) - hexagon(X_i)) \qquad \text{(Eq. 6.3)}$$

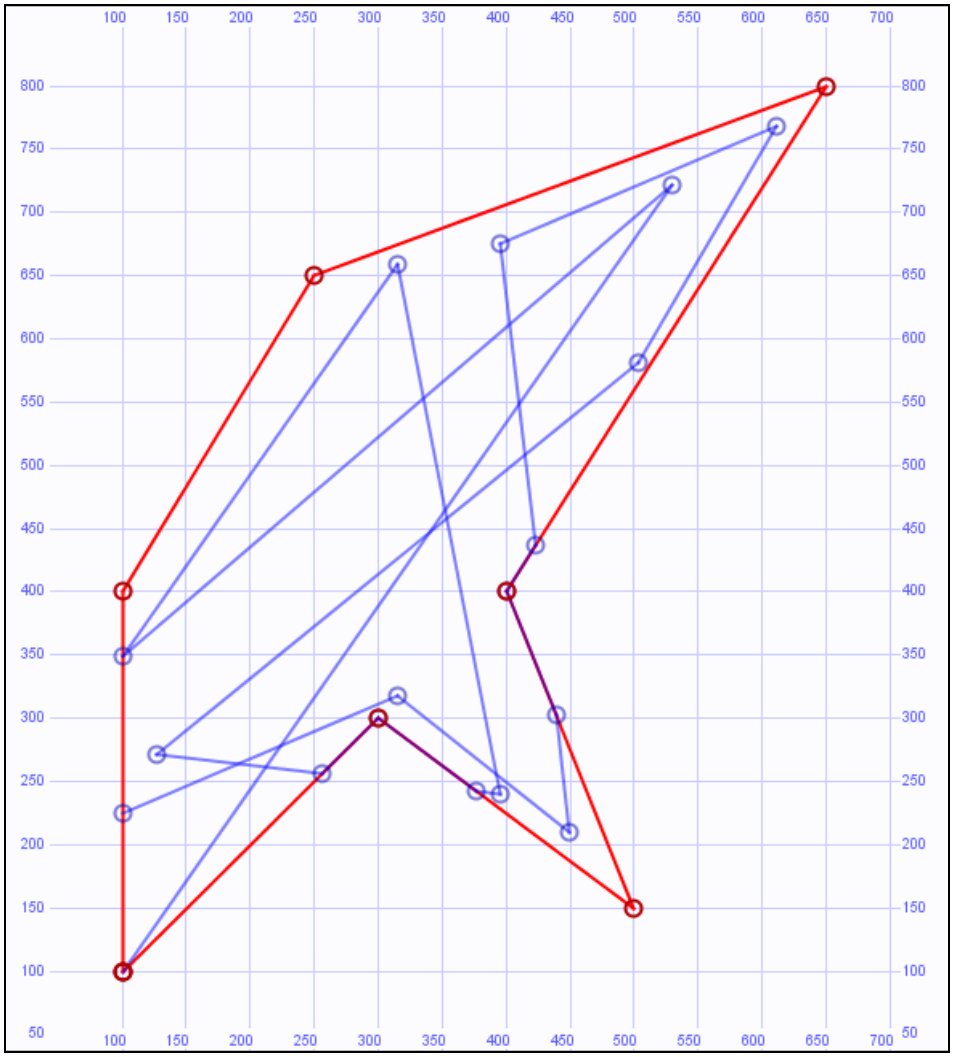


Figure 6.8. A Search Pattern for an Irregularly Shaped Operating Area Generated by an A-Star (A*) Search Biased Towards Patterns with Fewer Waypoints

Search patterns generated with this A* search tend to consist of legs that transit from one side of the area to the other in a rather disorganized manner. Additionally, they often repeatedly cover portions of the area as they traverse from side to side. This is counter to the implicit goal of generating a pattern that is more efficient

(i.e., shorter in length) than an adjusted predefined pattern. Rather, A*-generated plans are typically much longer than adjusted parallel-track patterns, and are only occasionally more efficient than adjusted square or rectangle patterns. Unfortunately, significant reduction in the value of k to favor more efficient search patterns quickly increases search-tree depth to an unacceptable degree.

c. Combined Best-First / A Based Search-Pattern Development*

In an attempt to overcome the shortcomings of this A* implementation, a modification to the basic A* algorithm is introduced. If the value of *k* is lowered (to 0.25 in this case), the cost of the partial path becomes more important than the estimated remaining cost when evaluating partial solutions. Thus, shorter partial patterns that make some progress towards the goal are favored over longer ones that make more progress. In order to minimize the time spent evaluating dead-ends, the A* agenda is pruned periodically using a best-first heuristic. After evaluating a predetermined number of candidates (five in the tested implementation), the search commits to the most promising partial plan and deletes all other candidates from the agenda. The A* search is restarted using the most promising partial plan as the start state. This process repeats until the goal is reached.

An important restriction during this search is that the ability to backtrack is limited. It is therefore crucial that the goal remain reachable from any state to ensure the discovery of a solution. The search graphs described here meet this requirement. Since the heuristic for determining potential successor states allows the partial pattern to be extended to any unvisited hexagon, it is evident that all hexagons in the search grid either have been or can be visited from any state in the graph. However, generated patterns may require clipping of legs that cross the concave portions of the polygon.

Search patterns generated by this combined best-first / A* algorithm, such as the one depicted in Figure 6.9, tend to be more efficient than those generated by the previously discussed A* search (a specific comparison of the various search-pattern planners is provided later in this section). Based on numerous test cases, the average total distance traveled in executing search plans generated by this algorithm are approximately 13 percent shorter than those generated by the A* planner. Unfortunately, these plans still are still less efficient than adjusted parallel-track patterns and only

improve on adjusted square patterns in six of ten test cases. Thus, they do not accomplish the goal of generating area-specific search plans that improve upon area-adjusted predefined patterns. Additionally, the average run time of the best-first / A* planner is over six times longer than the run time of the A* planner. From an implementation standpoint, however, this does not rule out the combined search. Since the search commits to promising partial plans early in the search process, the total planner run time is not as important as it is for planners that do not generate any usable results until they have run to completion.

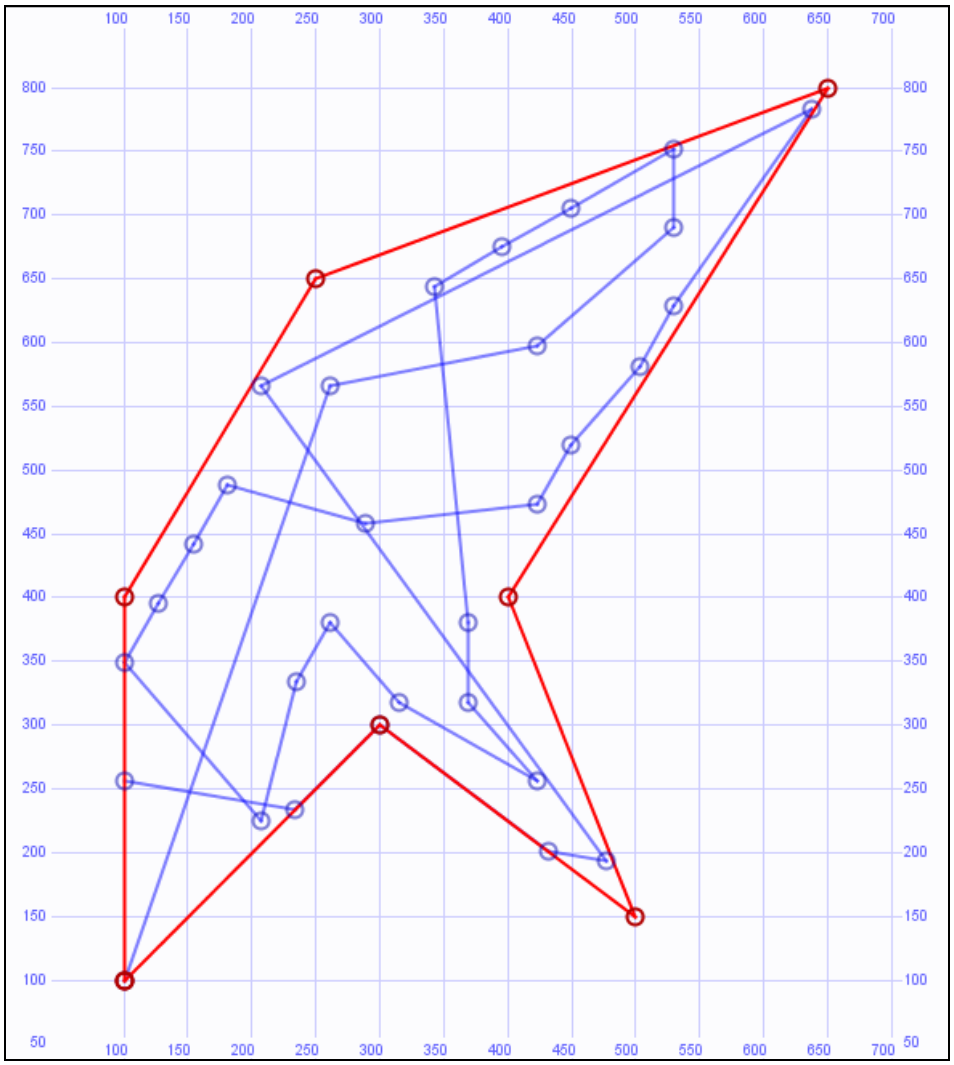


Figure 6.9. A Search Pattern for an Irregularly Shaped Operating Area Generated using a Combined Best-First / A* Search

d. Use of Hill-Climbing Search for Area Search Pattern Generation

Despite the improved efficiency of search patterns generated by a combined best-first / A* search, the example depicted in Figure 6.9 still leaves significant room for improvement. The improvement over the previous algorithm does, however, provide evidence that a pure best-first approach might provide still-better results. Utilizing Equations 7 and 8 with k set to 0.25 to favor shorter-path solutions, a hill-climbing search is easily implemented. An extension of the combined best-first / A* search, a hill-climbing search is achieved by clearing the search agenda of all but the most-promising partial plan at each iteration. As with the combined best-first / A* search, the lack of backtracking necessitates that the goal state be reachable from any state in the search graph.

In virtually every test case, the search pattern generated using this search was shorter in length than the pattern generated using the A* or combined best-first / A* search implementations (the average distance traveled by the test case patterns was 25 percent less than the A* average distance). Additionally, the average planner run times were almost three times faster than those of the combined best-first / A* planner (but almost twice as slow as the A* planner). Discouragingly, however, the efficiency of plans generated using this algorithm still do not consistently beat that of adjusted parallel-track patterns, averaging slightly longer patterns and actually improving on the adjusted parallel-track length in only one of ten test cases. The pattern depicted in Figure 6.10 provides an example of why this shortfall occurs. This plan covers a number areas multiple times, recrossing its own path on four occasions and passing within close proximity to previous legs on two more. Nevertheless, these patterns are nearly always shorter than adjusted expanding or shrinking-square patterns and frequently shorter than adjusted parallel-track patterns. This improvement coupled with the relative speed of planner execution make this hill-climbing implementation potentially useful.

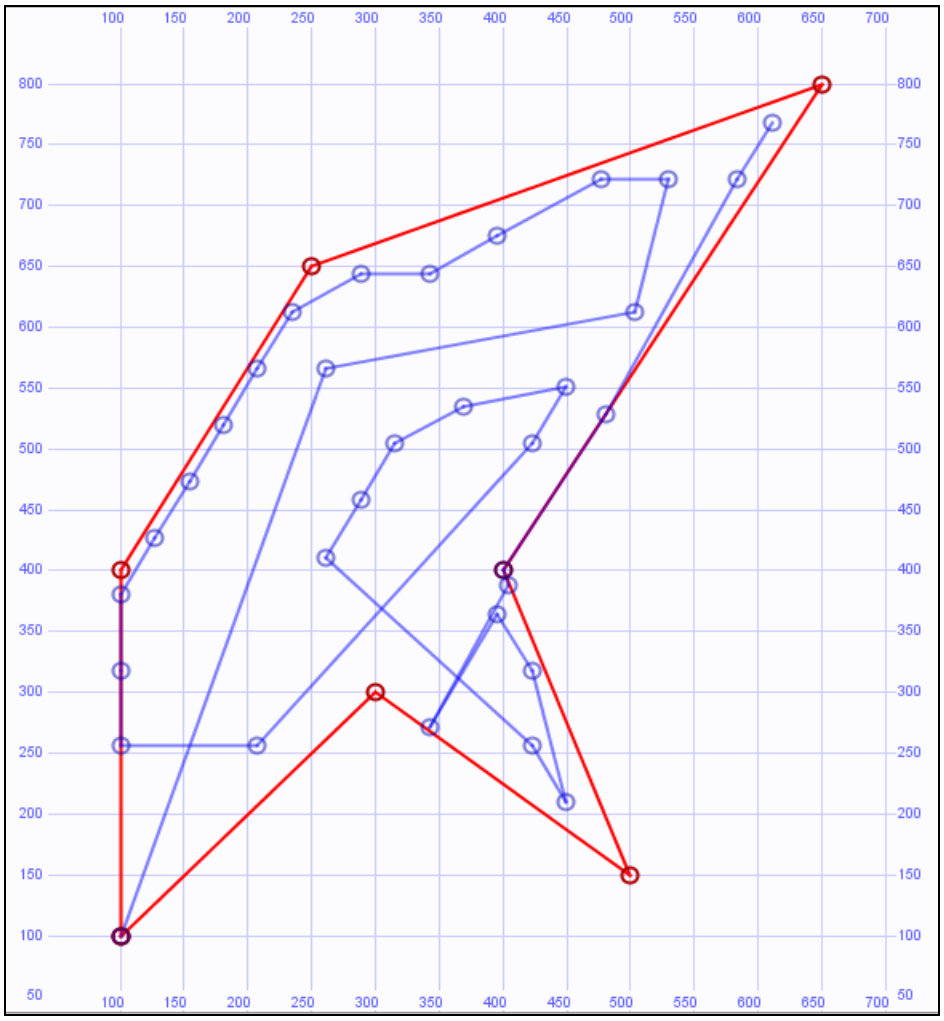


Figure 6.10. A Search Pattern for an Irregularly Shaped Operating Area Generated using a Hill-Climbing Search that does not Allow Backtracking

e. Search Pattern Development Using Iterative Improvement of a Traveling Salesman Problem Solution

The most successful methods of planner-based search pattern generation implemented in the course of this work treated the search as an adaptation of the traveling salesman problem. Although known to be of nondeterministic-polynomial-time-hard (NP-hard) complexity, it is possible to develop a "good" if not "optimal" solution much more quickly (Corman, et al., 90). Similar to the search-based pattern planners, the traveling-salesman-problem-based algorithm divides the search area into hexagons that are to be visited but, unlike the previous methods, the center of each hexagon must actually be used as a waypoint in the search pattern. This visitation criteria allows increasing hexagon diameter to match track spacing without sacrificing coverage. When

the iterative improvement algorithm (described in pseudocode in Figure 6.11) begins, the search points (i.e., the set of hexagon centers) are loaded into an array representing visitation order. The only stipulation is that the first element must be the search starting point (i.e., the area center for a point-focused search or the closest search point to the area entry otherwise). Each array element (with the exception of the first) is iteratively compared against other the elements. If the path defined by swapping the elements is shorter than the current path, the elements are swapped. This process is repeated until no further improvements are obtained.

```
Let searchPts =
    array of search area grid hexagon centers
Let startD = 0
Let endD   = pathDistance(searchPts)
While endD != startD
    For pt1Index = 1 to count(searchPts) - 1
        For pt2Index =
            pt1Index + 1 to count(searchPts)
            Let d1 = pathDistance(searchPts)
            swap(searchPts, pt1Index, pt2Index)
            Let d2 = pathDistance(searchPts)
            If d1 < d2
                swap(searchPts, pt1Index, pt2Index)
    startD = endD
    endD   = min(d1, d2)
```

Figure 6.11. Progressive Improvement of a Traveling Salesman Problem Solution to Generate Efficient Search Patterns for Arbitrarily Shaped Areas

Search patterns generated using this algorithm are easily the most efficient of all of the planner-generated patterns discussed thus far, averaging a 40 percent improvement over patterns generated by the A* search for the test cases. As the example of Figure 6.12 indicates, these patterns gravitated towards relatively regular patterns with few intersections. Despite a slight susceptibility to convergence on local minima (a common characteristic of iterative improvement algorithms), these patterns were also shorter than the adjusted parallel track pattern in seven of ten test cases. Finally, the run

time of the traveling-salesman-problem-based planner was at least an order of magnitude better than any of the previously discussed planners in all test cases.

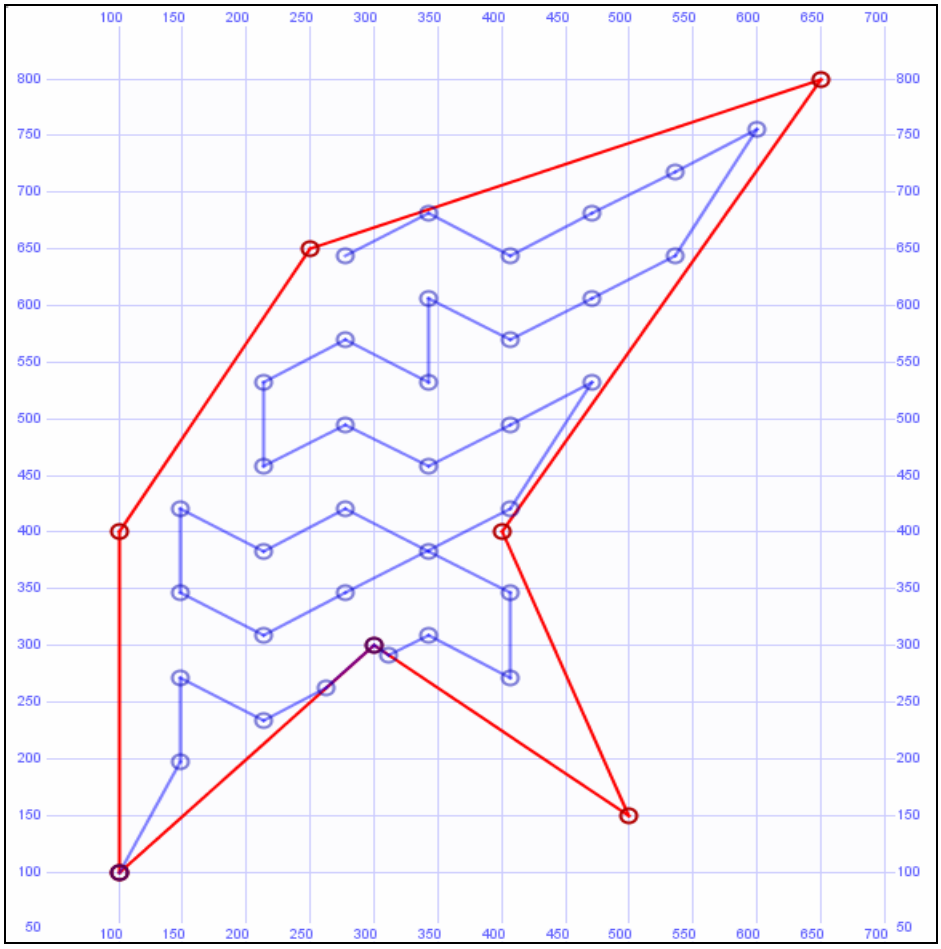


Figure 6.12. A Search Pattern for an Irregularly Shaped Operating Area Generated using the Traveling-Salesman-Problem-Based Algorithm of Figure 6.11

f. Iterative Improvement of Traveling Salesman Problem Search Patterns using Simulated Annealing

One pitfall of the algorithm of Figure 6.11 is the tendency to converge on local minima. In many instances, the initial traversal sequence contains loops and patterns that cannot be untangled by pair-wise exchange based solely on the relative quality of the pre- and post-switch patterns (the pattern depicted in Figure 6.12 contains one such loop). For this reason, a number of traveling salesman problem approaches incorporate random components that allows the search to explore the search space more freely. Among these are genetic algorithms and simulated annealing (Russell and Norvig, 03).

The algorithm of Figure 6.11 is easily augmented to incorporate simulated annealing by modifying the conditions upon which points in the sequence are switched. Equation 6.4 is used to determine a probability that two points *i* and *j* are switched even if the resulting path (p_{new}) is longer than the original one (p_{old}). The points are always switched it the resultant path is shorter. The variables *t* and t_{max} are the current and starting "temperatures" of the annealing system, respectively, and *k* is a weighting factor applied to the difference in path lengths (the best results were obtained using a value of 30). Cooling in the implemented system is linear to a minimum of zero (at which point the algorithm continues according to Figure 6.11). The equation makes seemingly less than optimal switches more likely (to a maximum probability of 0.5) while the system is still "warm" and if the switch results in a path that is only slightly longer than the original path.

$$P\left(switch_{i,j}\right) = \left(1 - \min\left(\frac{k\left(\left|p_{new}\right| - \left|p_{old}\right|\right)}{\left|p_{old}\right|}, 1.0\right)\right)\left(\frac{t}{2t_{\max}}\right) \qquad \text{(Eq. 6.4)}$$

Typical results of the simulated-annealing-based traveling salesman problem algorithm described above are along the lines of the search pattern depicted in Figure 6.13 which is substantially shorter than even the pattern generated by the previous traveling salesman problem iterative improvement algorithm. However, the stochastic nature of the selection makes results nondeterministic and similar performance is not guaranteed. For this reason, simulated-annealing implementations often use multiple searches to increase the likelihood of obtaining at least one near-optimal solution. This approach might be applicable for the pre-mission search pattern generation described in this chapter, but is probably impractical for on-vehicle control as described in Chapter VII. The usefulness of a planner such as this one for actual vehicle control depends on factors such as the likelihood of a suboptimal plan and the degree of potential suboptimality. These and other characteristics of the simulated-annealing traveling salesman problem iterative-improvement algorithm are discussed in the next section.

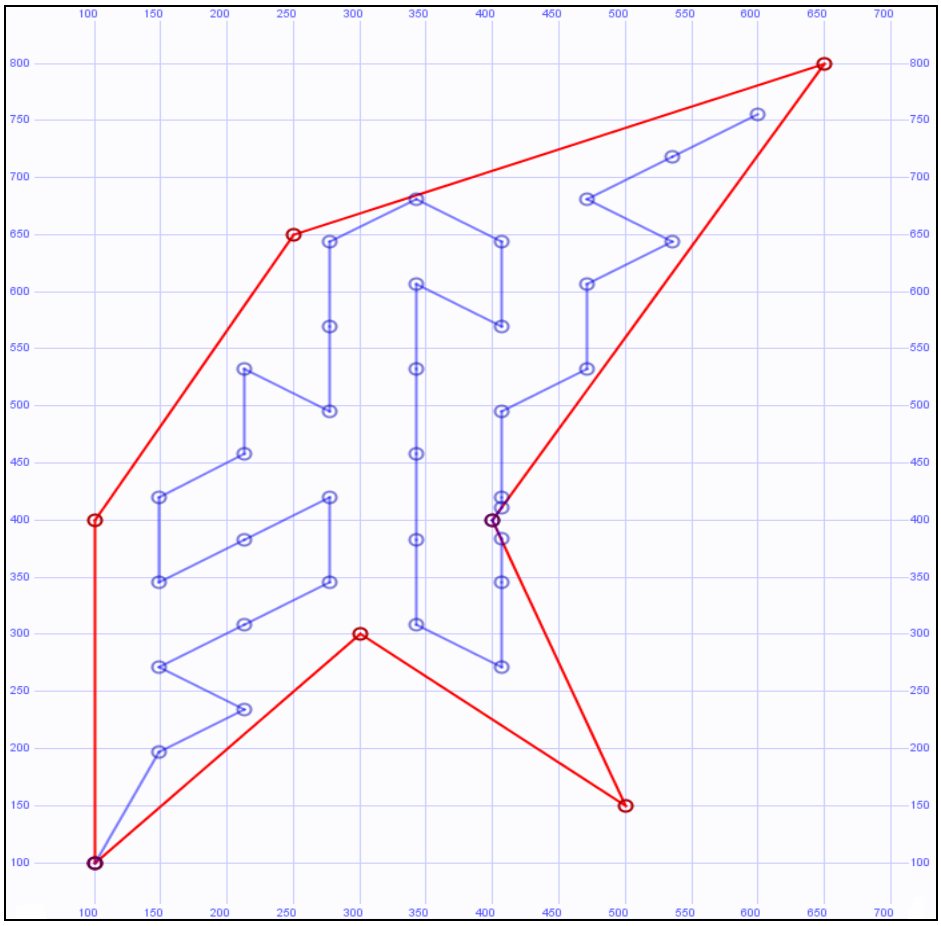


Figure 6.13. An Irregular Area Search Pattern Derived using Simulated-Annealing-Based Iterative Improvement of a Traveling Salesman Problem Solution

g. Comparing Automated Search Pattern Generation Techniques

From the preceding discussion, it is clear that the traveling-salesman-problem-based search pattern planners outperform the search-based planners in both the efficiency of the generated search plans and the execution speed of the planners themselves. Figures 6.14 through 6.16 provide a more detailed comparison of all five planners. Because of its nondeterministic nature, the average, minimum, and maximum values for 1000 runs of the simulated annealing planner are depicted. Where relevant, a comparison against expanding-square and parallel-track patterns is provided as well.

Figures 6.14 and 6.15 provide a comparison of the search-pattern length for both planner-generated and adjusted preplanned patterns for ten concave-polygonal areas along the lines of the one depicted in the previous examples. Figure 6.14 depicts absolute pattern length while Figure 6.15 depicts normalized length using the area-

adjusted parallel-track pattern as the baseline. These graphs quantify the previous observations concerning the relative efficiency various planner-generated plans. The traveling-salesman-problem-based planners are the only ones that produced better results than the area-adjusted parallel-track pattern on a reasonably consistent basis. The simulated annealing planner, in particular, performed well, providing the best solution in eight of ten test cases. In fact, even the worst-case simulated-annealing results improved upon the area-adjusted parallel track pattern in half of the test cases. The traveling-salesman-problem-based and hill-climbing planners were all able to generate patterns of shorter length than the area-adjusted expanding or shrinking-square or rectangle patterns.

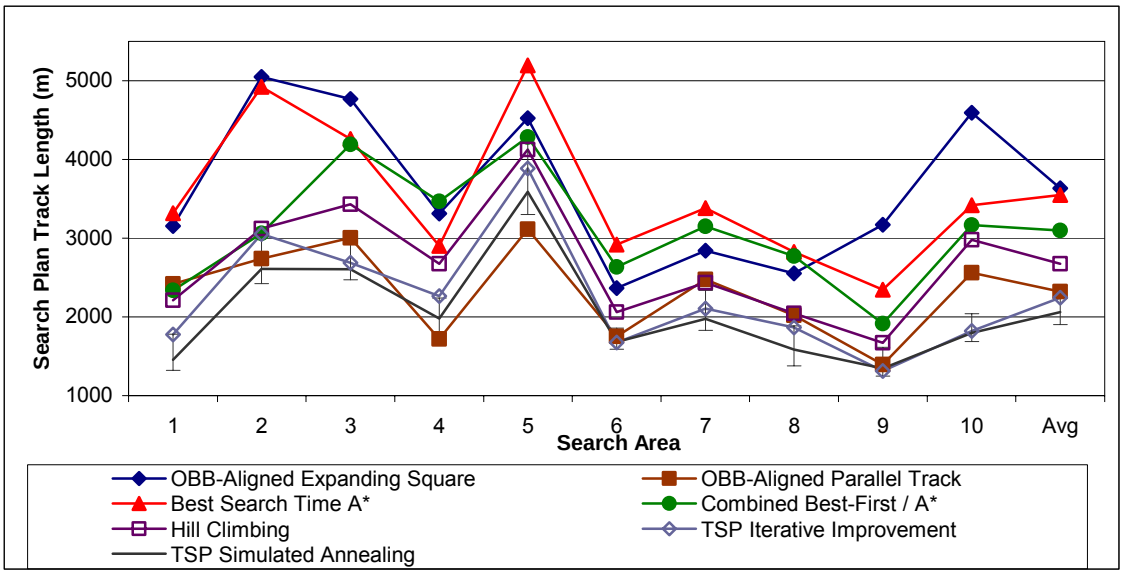


Figure 6.14. A Comparison of Absolute Track Length of Planner-Generated and Area-Adjusted Preplanned Search Patterns for Concave-Polygonal Areas

Figure 6.16 shows the planner run times in generating plans for the same ten concave-polygonal areas. The run times of the combined best-first / A* planner are significantly longer than the next slowest planner. Most striking, however, is the degree to which traveling-salesman-problem-based planners outperform the search-based planners. Ultimately this is not surprising since the computational complexity of Figure 6.11's algorithm is $O(n^2 m)$ where n is the number of search points and m is the number of iterations that continue to show improvement. This is not insubstantial by any means, but it is significantly less complex than the potential exponential complexity of the A*

search. Despite the fact that the combined best-first / A* and hill-climbing searches significantly reduce complexity by eliminating most search-tree branches or minimizing search-tree depth, Figure 6.16 provides an indication that their complexity as implemented still exceeds that of a low-order polynomial algorithm by a great deal. The increase in run time of the simulated annealing planner over that of the basic traveling-salesman-problem-based planner results from the continuation of the algorithm until the system temperature reaches zero

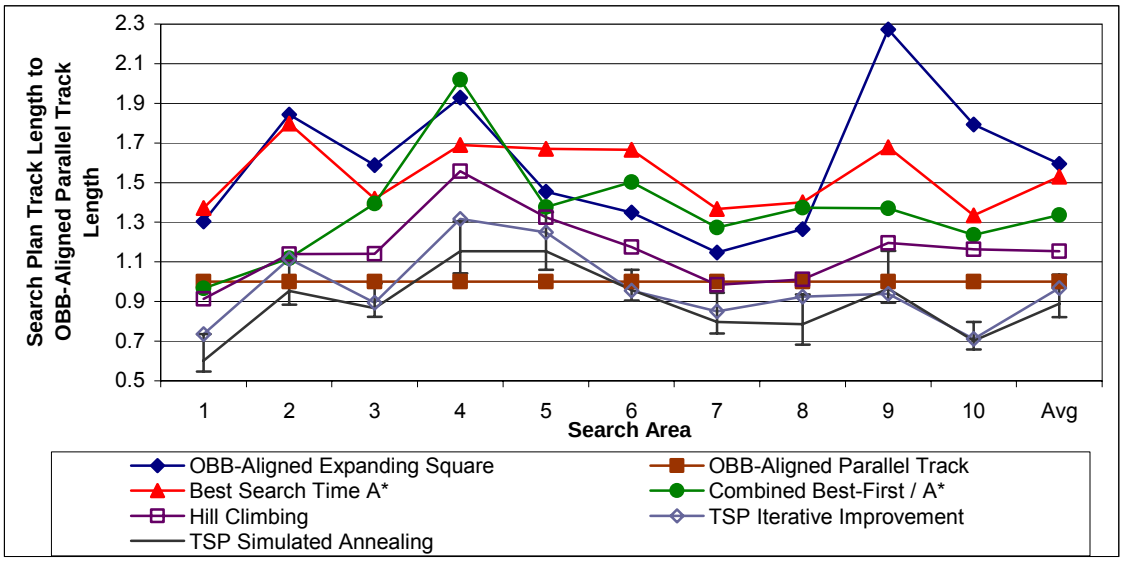


Figure 6.15. A Comparison of Normalized Track Length of Planner-Generated and Area-Adjusted Preplanned Search Patterns for Concave-Polygonal Areas

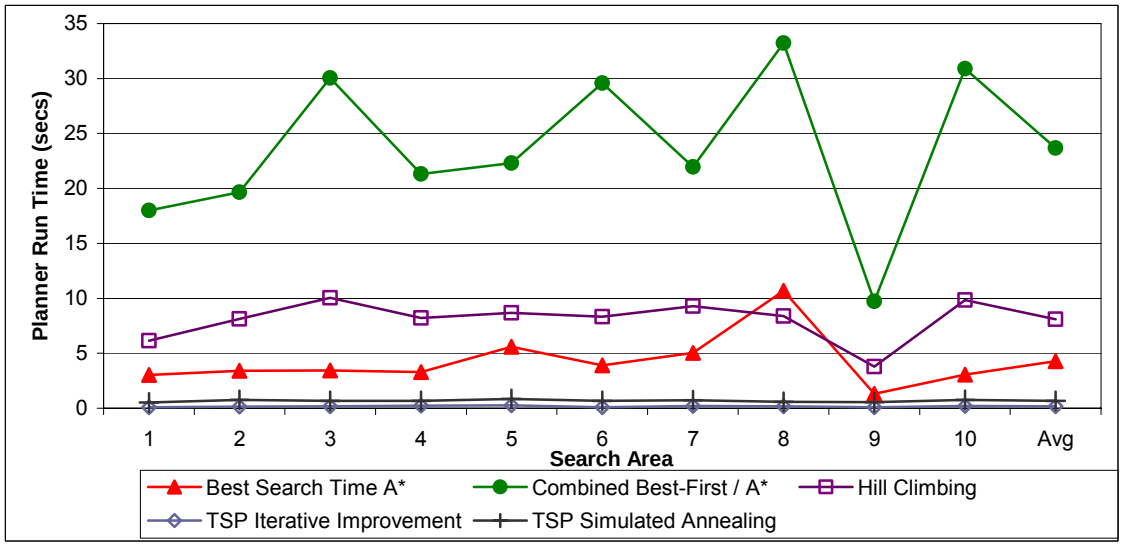


Figure 6.16. Comparison of Run Times for Search Pattern Planners

The data depicted in Figures 6.14 through 6.16 provide evidence to support the use of the traveling-salesman-problem-based planners for the generation of search plans for irregularly shaped operating areas. Further, the observation that the worst-case solutions generated through simulated annealing are both rare (i.e., the average solution is significantly closer to the best-case solution than the worst-case one) and on par with basic traveling salesman problem solutions indicates that the this planner is suitable not only for pre-mission use, but for on-vehicle use as well.

It is worth noting that operating areas of this sort (i.e., irregular-polygonal areas) are the exception rather than the rule. The majority of search areas can be classified as circles, rectangles or concave polygons. For these area types, preplanned patterns still provide the most intuitive and straightforward coverage.

4. Global Path Planning in Script Generation

Returning to the original motivations for this chapter, the second requirement for converting a declarative agenda to a task-level behavior script is global path planning between operating areas. For pre-mission planning purposes, this amounts to planning a path from either the specified launch position (for the first goal in the mission) or the last geographic position in the sub-script corresponding to a particular goal to the closest point in operating area of the next goal to be attempted or to the recovery position if the mission is complete.

Much recent research in the area of mobile robot path planning has focused on searches along the lines of the D* search. This search algorithm is similar to the A* algorithm in its use of actual partial path cost and estimated remaining path cost functions. It is often considered more suitable for real-time path planning, however, because it allows run-time modification of the incurred cost and estimated remaining cost for individual states as the search progresses. This makes the D* algorithm inherently more adaptable to the dynamic environments in which autonomous vehicles operate than many competing algorithms. (Ferguson and Stentz, 05)

As the proposed common autonomous vehicle data model evolves, it is likely that it will take on a more dynamic aspect. For the present, however, it does not include features that make D* inherently more desirable than other algorithms. In particular,

where pre-mission use is concerned, any algorithm features specifically supporting dynamic situations are superfluous (this is no longer the case when the planner is used for real-time vehicle control as described in Chapter VII). In addition, from the standpoint of the currently implemented common data model, path planning between operating areas is not dynamic in nature. All that is required is to plan a path that does not enter any avoid areas (i.e., the possibly dynamic cost of traversing a particular region is not relevant). Localized path planning and obstacle avoidance between global waypoints is conducted at a lower level of control than is currently addressed by the AVCL agenda. For these reasons, a fairly simple best-first search is used for path planning between operating areas.

Like operating areas, avoid areas are defined in an AVCL agenda using points, circles, rectangles or polygons. The algorithm begins with an agenda containing only a candidate path directly from the start to the goal. At each iteration of the search, the shortest candidate path in the agenda is tested. If this path does not enter any avoid areas, the best path has been discovered. If it does intrude into one or more avoid areas, it is removed from the agenda and two new candidate paths are added for each avoid area that the path enters. New candidate paths are generated by removing the offending path segment and replacing it with two segments: one from the start of the original segment to a tangent point on the edge of the avoid area and one from the tangent point to the original segment's end. A simple example is provided in Figure 6.17. The initial candidate intrudes into two avoid areas, so four new candidate paths are added to the agenda. In the second iteration, the shortest candidate enters one avoid area, so two more candidates paths are added to the agenda, one of which avoids all prohibited areas. Since there is a candidate path that is shorter than the newly generated viable plan, the ultimate solution is not encountered until the fourth iteration of the search.

Despite its simplicity, this search works well in the role of pre-mission path planning. Additionally, since an AVCL agenda essentially divides the world into two types of regions (i.e., those that the vehicle is to avoid and those that the vehicle can enter), it can also be applied to run-time path planning at the agenda level (i.e., global path planning between operating areas). Since the candidate path-selection criteria is equivalent to an A* search with a constant estimated remaining cost of zero and partial-

path cost equal to the path's length, the search can be guaranteed to find the shortest path from the start to the goal. As the robustness of AVCL agenda descriptions increases and become more dynamic in nature, it is likely that this search will be less useful and other search algorithms along the lines of D* will become more attractive.

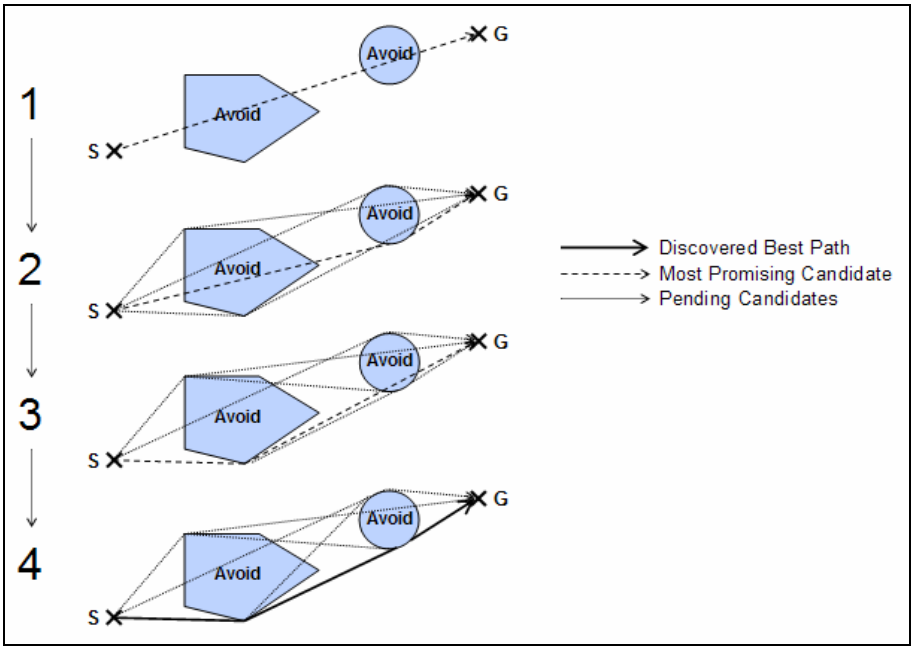


Figure 6.17. A Global Path-Planning Example using a Best-First Search to Discover the Shortest Path from Start (S) to Goal (G) that Bypasses all Avoid Areas

C. INFERENCE OF DECLARATIVE MISSION GOALS FROM TASK-LEVEL SCRIPTS

1. Overview

A second possible application of declarative missions is their generation from scripts. Superficially, it might seem of limited use to convert an existing task-level behavior script to a declarative agenda, however this capability does have potentially important applications. If autonomous vehicles are to operate as an integral part of a larger command and control system, they will be required to exchange information with systems outside of the autonomous vehicle domain. In most instances, the larger intent of a mission is of more use to command and control systems than a sequence of low-level commands. To accomplish this level of data exchange, methodologies must be developed to reasonably infer the intent of a series of task-level behaviors and to translate that intent into a declarative agenda. Additionally, this ability can improve data analysis

of archived missions by providing multiple levels of interpretation of individual missions. This approach is particularly advantageous if the semantic richness of the autonomous vehicle data model described here is extended into a more robust ontology, making the techniques appropriate for use with Semantic Web applicable. (Daconta, et al., 03).

The process for developing an appropriate declarative goal for a sequence of task-level behaviors consists of two parts. The most difficult is the inference of the appropriate goal type. Once a goal type has been selected, the actual derivation of goal parameters and the generation of the AVCL goal element itself is straightforward.

Three important assumptions are made when inferring goal types from task-level behavior scripts. The first is that the script corresponds to a single goal (and the transit to and from the operating area) as opposed to a series of goals, so the entire script is considered at once. This assumption is made primarily to simplify the implementation requirements, but the methods discussed in the following sections can be applied piecemeal to behavior sub-scripts if translation to a series of goals rather than a single goal is desired. The second assumption is that the task-level behavior sequence can be translated into a goal that is successfully completed. Finally, no attempt is made to infer the presence or location of avoid areas. It is assumed that the task-level behavior script calls for a transit to directly to and from the operating area. As with the single-goal assumption, this is intended to simplify the overall implementation and does not prevent future inclusion of sub-script-level analysis that considers the possibility of indirect transit between operating areas.

Two further points concerning goal-type inference bear mentioning. The first is that the process can be greatly simplified if the task-level behavior script is annotated using MetaCommand behaviors. However, the methods described here are intended to work with annotated or unannotated scripts, so they do not rely MetaCommand information to infer goal-types. Nevertheless, the presence of MetaCommand behaviors can improve the performance of these methods. Ultimately, the development of a more robust ontology can enhance the ability of scripts to be self-documenting and further simplify their interpretation.

The second point relates to the fact that the task-level behavior set in its current form does not fully implement all of the available AVCL goal types. There is no requirement to attempt to infer goal types that cannot be expressed as task-level behavior scripts. The implementations developed in this work, therefore, classify scripts as one of five possible goal types. The script classifiers will conclude that the most likely intent of a mission is to perform a point-focused search, perform an area search, patrol an area, monitor some sort of activity or transmissions, or simply transit to a new location.

2. Mission Goal-Type Inference using Case-Based Reasoning

Two methods were developed for inferring the intent of a series of task-level scripts. The first is case-based reasoning, a fairly simple form of machine learning that compares the characteristics of an instance of unknown classification against the characteristics of a set of known instances. The unknown case is classified based on the classifications of the most closely matching known instances (Mitchell, 97).

The case-based reasoning system implemented for script classification computes values for the 15 characteristics described in Figure 6.18. Each is defined on an inclusive range of zero to one and is assigned a weight, or importance as well. The "distance" of an unknown instance (x) from a known instance (r) from the set of known recall cases (R) is computed using Equation 6.5 where w_i is the weight assigned to characteristic i (individual weights are listed in Table 6.1), c_i is the function determining the characteristic value. The recall set consists of 75 "typical" task-level behavior scripts divided more or less equally between the potential vehicle and goal types.

$$distance(r,x) = \sum_{i=1}^{15} w_i \left| c_i(x) - c_i(r) \right| \qquad \text{(Eq. 6.5)}$$

Among the advantages of case-based reasoning are its simplicity and its intuitive nature—a specific classification is easily justified by noting its similarity to a known instance. Case-based reasoning is also more likely to work reasonably well when a limited number of recall cases are available than are many other classification methods (provided the set of recall cases adequately spans the domain). Further, "training" a case-based reasoning system is easily accomplished by adding instances that are incorrectly classified to the recall set along with their correct classification.

Stationary commands. This characteristic is a measure of how much the script calls for a stationary vehicle and is computed as the ratio of stationary behaviors to total behaviors. Behaviors meeting this criteria include Loiter and Hover behaviors, zero-value MakeKnots or MakeSpeed behaviors and Waypoint behaviors that are followed by a Wait behavior.

Area coverage. This characteristic is measures how thoroughly the track defined by the script covers the convex polygon containing the track. It is computed as the ratio of swept area (track length times sweep width minus intersections and out-of-area coverage) to convex polygon area.

Point focus. This characteristic is a measure of how much the mission focuses on a single point close to the center of the area. It is calculated by determining the maximum number of times a single point is visited and multiplying by a distance-from-area-center factor.

Parallel tracks. This characteristic measures the degree of parallelism in the legs of the mission. It is computed as the ratio of the number of legs for which at least one of the next three legs is parallel (within five degrees) to the total number of legs in the mission

Orthogonal tracks. Similar to the parallel tracks characteristic, this characteristic is computed as the ratio of 90 degree (within five degrees) turns in the mission to the total number of turns.

Multiple point visits. This characteristic is a measure of how many of the points defined in the mission are used more than once. It is computed as the ratio of points that are used more than once to the number of unique points used in the mission.

Local finish. This characteristic is a measure of the proximity of the final waypoint to the launch position, adjusted for the type of vehicle.

Number of legs. This characteristic is a measure of the number legs in the mission and is computed as the reciprocal of the number of waypoints.

Sweep width ratio. This characteristic is a measure of how small the operating area is relative to the sensor sweep width. It is computed as the ratio of the stationary sweep area (the area of a circle of sweep width radius) to the area of the convex polygon containing the track.

Center start. This characteristic measures how quickly the script orders the vehicle to the center of the operating area. It is computed based on the proximity of the nearest of the first three mission waypoints to the center.

Fixed vertical. Only relevant for UUV and UAV missions, this characteristic is a measure of the degree to which the script calls for a fixed depth or altitude. It is computed as the ratio of commands that call for a depth or altitude change to commands that can conceivably call for a change in depth or altitude.

On surface. Relevant only for UUV missions, this characteristic is a measure of the amount of time the vehicle spends on the surface and is computed as the ratio of the commands that call for the vehicle to be on the surface to commands that are capable of specifying depth.

Uses search speed. Based on default search speed, this characteristic is computed as the ratio of behaviors calling for the search speed (within 10 percent of maximum vehicle speed) to speed-ordering behaviors.

Uses patrol speed. This characteristic is computed identically to the uses-search-speed characteristic but uses the vehicle's default patrol speed (which may or may not be the same as the default search speed).

Uses transit speed. Again, this characteristic is computed in the same manner as the uses-patrol-speed and uses-search-speed characteristics, but uses the vehicle's default speed for transit between operating areas.

Figure 6.18. Properties used to Classify AVCL Task-Level Behavior Scripts using Case-Based Reasoning

Characteristic	Weight	Characteristic	Weight	Characteristic	Weight
Stationary Commands	1.00	Multiple Point Visits	0.25	Fixed Vertical	1.00
Area Coverage	1.00	Local Finish	1.00	On Surface	0.20
Point Focus	0.50	Number of Legs	1.20	Uses Search Speed	0.20
Parallel Tracks	1.00	Sweep Width Ratio	0.50	Uses Patrol Speed	0.20
Orthogonal Tracks	1.00	Center Start	1.00	Uses Transit Speed	0.20

Table 6.1. Characteristic Weights for Case-Based Reasoning Classification of Task-Level Behavior Scripts

Disadvantages of case-based reasoning include relative inefficiency arising from the need to compare unknown instances against each recall case. Also an issue is the potential for a poor recall set (i.e., one that does not accurately reflect the population at large) to improperly bias classifications or to blanket the search space in such a way that they are more or less random. This is especially an issue if some or all of the characteristic values or weights are poorly chosen. (Mitchell, 97)

3. Mission Goal-Type Inference using Naïve Bayes Reasoning

The characteristics used by the case-based reasoning system for task-level-script classification are also potentially useful with other machine-learning methods. A neural network or support-vector machine, for instance, might classify unknown instances based on the same characteristics. Also potentially applicable are probabilistic techniques. These typically determine a maximum *a posteriori* hypothesis for unknown instance classification based on the probabilities of the instance's characteristics. Among the most common probabilistic learning methods are those relying on conditional probabilities and Bayes Theorem to compare the likelihood of all potential hypotheses. The general form of the equation for the probability of a hypothesis (H) given observed characteristics (c_1 through c_i) is given by Equation 6.6. Unfortunately, the conditional probabilities on the right-hand side of the equation can be difficult to determine, so direct use of Equation 6.6 is often impractical. In cases where the values of some characteristics are influenced by the values of others (i.e., some characteristics are dependent on others), a common approach is to develop a Bayesian network reflecting the various dependencies. If, on the other hand, if mutual independence of the characteristic values can be safely assumed, the equation can be rewritten as Equation 6.7. Values for the individual probabilities in this equation are derived from the set of known instances, making Equation 6.7 easy to apply

in practice. Referred to as naïve Bayes (because of the probably naïve assumption that all characteristic values are independent), this form of probabilistic learning was implemented for the purpose of classifying task-level behavior scripts.

$$P(H \mid c_1 \ldots c_i) = \frac{P(H) P(c_1 \ldots c_i \mid H)}{P(c_1 \ldots c_i)} \quad \text{(Eq. 6.6)}$$

$$P(H \mid c_1 \ldots c_i) = \frac{P(H) \cdot P(c_1 \mid H) \cdot \ldots \cdot P(c_i \mid H)}{P(c_1) \cdot \ldots \cdot P(c_i)} \quad \text{(Eq. 6.7)}$$

Unlike the characteristics used by the case-based reasoning implementation, the characteristic values of Equation 6.7 are discrete, so the characteristics used by the case-based reasoning system require modification. The 12 characteristics used in the naïve Bayes implementation are described in Figure 6.19 and the associated conditional and unconditional probabilities are listed in Table 6.2. These characteristics were chosen because of their relative independence as well as the potential that their values might be indicative of the script's intent.

The computational overhead of a probabilistic learning system along the lines of the naïve Bayes classifier is significantly reduced over that of a case-based reasoning system since the characteristics of individual recall instances are not required at run time (they are implicitly captured by the probabilities used in the equations). Additionally, the influence of individual characteristics on the outcome is essentially self adjusting in that the probabilities associated with characteristics that are not true solution indicators tend to be the same for all potential hypotheses. This eliminates the need for the somewhat arbitrary process of tuning the weight applied to each characteristic. It does not, however, alleviate the requirement to determine and calculate suitable characteristics.

Among the most significant disadvantages of probabilistic machine learning in this application is a potentially insufficient number of known instances from which to derive probabilities. In addition to the derivation of potentially invalid probabilities, the use of too few test cases can result in probabilities of 1.0 or 0.0 that can unacceptably bias the solution based on a single characteristic value. In order to minimize this disadvantage, the probabilities in Table 6.2 were computed using all of the 104 available test cases, rather than just the 75 used in the case-based reasoning system. Unfortunately,

even the use of all available test cases does not eliminate probabilities of 1.0 or 0.0. In order to prevent these values from exerting too much influence on the solution, these probabilities were manually adjusted to values that are considered more realistic (indicated with italics in Table 6.2). Additionally, useful data concerning the unconditional probabilities of the individual hypotheses is hard to come by. In fact, it is likely that these probabilities are dependent on the overall role of the vehicle in question (e.g., military, industrial, scientific, etc.). The current naïve Bayes implementation makes the assumption that all hypotheses are equally likely, effectively basing the ultimate classification solely on the conditional probabilities of the characteristics.

Characteristic (ci)	Monitor Transmissions		Patrol		Reposition	
	P(ci|H)	P(~ci|H)	P(ci|H)	P(~ci|H)	P(ci|H)	P(~ci|H)
Stationary Commands	0.8462	0.1538	*0.0500*	*0.9500*	*0.0500*	*0.9500*
Area Coverage	0.0769	0.9231	0.8421	0.1579	0.5000	0.5000
Point Focus	*0.0500*	*0.9500*	0.0526	0.9474	*0.0500*	*0.9500*
Parallel Tracks	*0.0500*	*0.9500*	0.6842	0.3158	*0.0500*	*0.9500*
Orthogonal Tracks	0.1538	0.8462	0.8421	0.1579	0.1667	0.8333
Multiple Point Visits	*0.0200*	*0.9800*	0.8421	0.1579	*0.0200*	*0.9800*
Local Finish	0.8462	0.1538	0.7895	0.2105	*0.0010*	*0.9990*
Sweep Width Ratio	0.7692	0.2308	0.1579	0.8421	0.7500	0.2500
Number of Legs	*0.0750*	*0.9250*	0.8947	0.1053	*0.0750*	*0.9250*
Changes Speed	0.5385	0.4615	*0.0500*	*0.9500*	0.1667	0.8333
Center Start	*0.0500*	*0.9500*	0.1053	0.8947	0.0833	0.9167
Has Sectors	*0.0250*	*0.9750*	*0.1000*	*0.9000*	*0.0250*	*0.9750*

Characteristic (ci)	Area Search		Point Search		P(ci)
	P(ci|H)	P(~ci|H)	P(ci|H)	P(~ci|H)	
Stationary Commands	*0.0500*	*0.9500*	*0.0500*	*0.9500*	0.1183
Area Coverage	0.7000	0.3000	0.8421	0.1579	0.6452
Point Focus	*0.0500*	*0.9500*	0.1579	0.8421	0.0430
Parallel Tracks	0.5000	0.5000	0.6842	0.3158	0.4409
Orthogonal Tracks	0.5333	0.4667	0.3684	0.6316	0.4624
Multiple Point Visits	0.1000	0.9000	0.1579	0.8421	0.2366
Local Finish	0.6000	0.4000	0.7368	0.2632	0.6237
Sweep Width Ratio	0.0333	0.9667	*0.0500*	*0.9500*	0.2473
Number of Legs	0.8667	0.1333	0.9474	0.0526	0.6559
Changes Speed	*0.0250*	*0.9750*	*0.0250*	*0.9750*	0.0968
Center Start	0.0667	0.9333	0.5789	0.4211	0.1613
Has Sectors	*0.0500*	*0.9500*	0.2105	0.7895	0.0430

Table 6.2. Probabilities Used in the Naïve Bayes Classification of AVCL Task-Level Behavior Scripts (Italics Indicate Probabilities that were Manually Adjusted from Computed Values of 0.0 or 1.0)

Stationary commands. Similar to the characteristic used in the case-based reasoning system, this characteristic has a value of "true" if the ratio of behaviors that command the vehicle to remain stationary to behaviors capable of ordering a reposition is greater than 0.33.

Area coverage. This characteristic is computed as in the case-based-reasoning system. It is considered "true" if the area swept by the mission track (minus intersections and out-of-area sweep) is greater than 75 percent of the operating area.

Point focus. This characteristic is uses the same formula as in the case-based reasoning system and has a value of "true" if the computed value exceeds 0.2.

Parallel tracks. This characteristic is based on the degree to which the mission has parallel tracks. It has a value of "true" if at least 45 percent of the legs are parallel (within five degrees) to at least one of the next three legs.

Orthogonal tracks. Like the characteristic of the case-based-reasoning system, this characteristic is based on the vehicle turns commanded by the behavior script. If at least 25 percent of the turns commanded by the script are approximately 90 degrees, then the characteristic has a value of "true."

Multiple point visits. This characteristic measures the number of mission points that are visited more than once over the course of the mission. If at least 25 percent of the mission's waypoints are revisited, a value of "true" is assigned.

Local finish. Similar to the corresponding characteristic in the case-based reasoning system, this characteristic has a value of "true" if the task-level behavior script calls for launch and recovery at approximately the same position. The distance threshold upon which the characteristic is based is dependent on the vehicle type.

Number of legs. This characteristic has a value of "true" if the mission script defines 10 or more legs not including transits to and from the operating area.

Sweep width ratio. As with the same characteristic in the case-based reasoning system, the sweep width ratio is a measure of how small the operating area is relative to the vehicle's sensor sweep width. It has a value of "true" if a stationary vehicle is capable of sweeping at least 75 percent of the area.

Speed changes. This characteristic is a measure of how fixed the ordered vehicle speed is over the course of the mission. If at least 33 percent of the behaviors capable of ordering a speed change do so, the characteristic has a value of "true."

Center start. This characteristic has a value of "true" if one of the first three in-area waypoints is close to the center of the convex polygon defining the operating area.

Has sectors. The only naïve Bayes classifier characteristic that does not correlate to any characteristic of the case-based reasoning system, this characteristic measures how much the mission path forms sectors focused on the center of the operating area. It has a value of "true" if the ratio of sectors (i.e., three or four leg sequences that start and end near the operating area center) to the number of mission legs exceeds 0.15.

Figure 6.19. Boolean Characteristics used for Naïve Bayes Classification of AVCL Task-Level Behavior Scripts

4. Comparing the Performance of the Case-Based Reasoning and Naïve Bayes Script Classifiers

Testing of both the case-based reasoning and naïve Bayes script-classification systems was conducted using 104 test missions with known classifications. In addition to the ideal classification, some test missions were assigned a classification that was considered acceptable but not ideal. For instance, it might be acceptable to mistake an area-search script for a patrol script. The test mission set included the missions upon which the system is based, however no mission was used in its own classification, so this was not allowed to bias the results. That is, individual test cases were excluded from the case-based reasoning recall set and naïve Bayes probabilities were recomputed prior to each classification without the test case. Performance for individual goal-types is provided in Figure 6.20 in the form of precision (the percentage of missions identified as a particular type that actually are that type) and recall (the percentage of missions of a given type that were correctly identified) for both systems. Separate data points are provided for ideal and acceptable classifications.

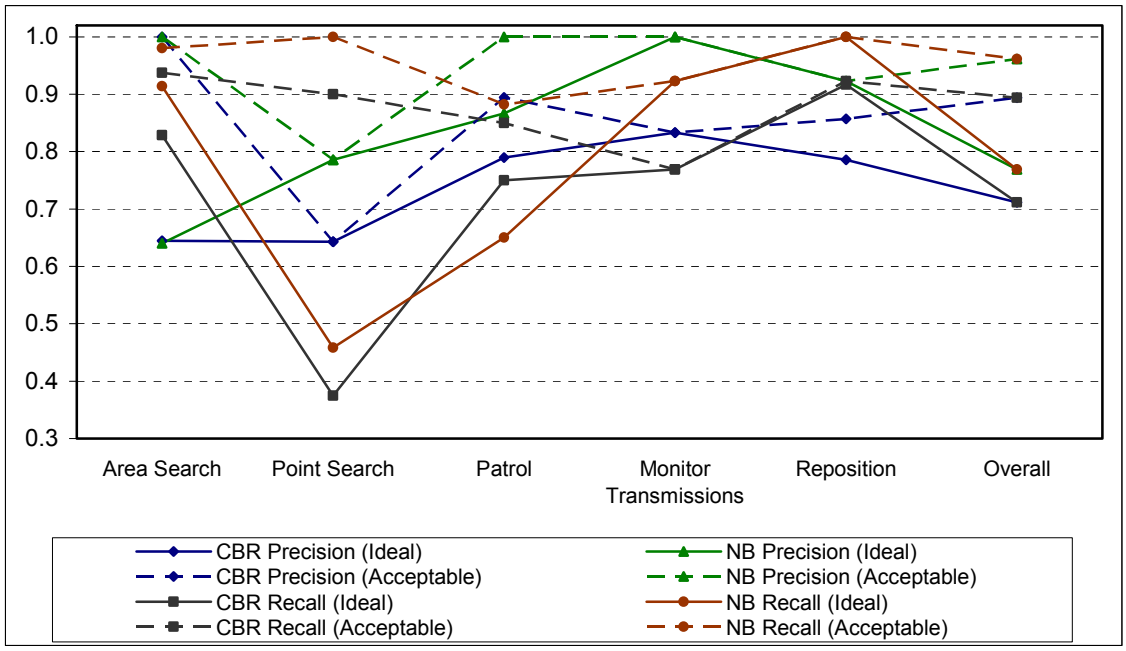


Figure 6.20. A Comparison of Individual Goal-Type Performance of the Case-Based Reasoning and Naïve Bayes Task-Level Behavior Script Classifiers

Overall, case-based reasoning system performance was encouraging, with 71 of 104 test scripts receiving their ideal classification. An additional 20 scripts were

acceptably, if not ideally, classified. Both precision and recall for most goal types exceeded 0.75 when using acceptability as the criteria. However, the drop-off in precision for point-search scripts was significant, with only 65 percent of the scripts classified as point-searches actually falling into this category. Potential causes for this anomaly include the absence of characteristics that are truly indicative of point searches and less-than-ideal weighting of characteristics.

When compared to the case-based reasoning classifier, the naïve Bayes system performed surprisingly well despite the admittedly small training set. In fact, in virtually every category both precision and recall using naïve Bayes analysis exceeded that of the case-based reasoning classifier. In total, 80 of 104 test scripts were assigned their ideal classification, and an additional 20 were classified acceptably for an overall precision of 0.96. As depicted in Figure 6.20, both precision and recall of the naïve Bayes classifier exceeded 0.75 for all categories and exceeded 0.85 in every case except point-focused-search script precision.

Both the case-based reasoning and naïve Bayes implementations provide promising enough results to merit further investigation. It is likely that overall performance of these particular systems can be improved by identifying the truly relevant characteristics and appropriate weights through a more rigorous characteristic versus goal type analysis, and increasing the size of the training set to more accurately represent the problem space. Additionally, the investigation of case-based reasoning selection methods along the lines of *k*-nearest neighbor, more robust probabilistic learning methods (i.e., ones that do not rely upon the assumed independence of the components), and other data-mining and machine-learning methods might also prove worthwhile. Finally, figure 6.20 clearly indicates that the performance of both systems varies with the type of mission being classified. This implies that analysis of individual goal-type classification using a mechanism such as receiver operating characteristics curves (Montgomery and Runger, 03) might prove useful in fine tuning both systems to improve performance.

D. SUMMARY

This chapter has described potential off-vehicle uses for the declarative mission portion of the proposed common autonomous vehicle data model. Of these, the most obviously applicable is the conversion of declarative agendas into task-level behavior

scripts that can be loaded into vehicles for execution. These conversions rely on Boolean propositions based on the characteristics of the goals and the operating area, decision trees, predefined waypoint-pattern templates, and in some cases artificial intelligence planners to generate task-level behavior scripts. Limitations arising from the static nature of scripts notwithstanding, this methodology can be used to convert most agendas into task-level behavior scripts that will closely resemble manually developed scripts.

The second off-vehicle application of declarative agendas is the reverse conversion—that is, converting a task-level behavior script into an agenda. This sort of translation is potentially important if autonomous vehicle systems are to interact with external command and control systems. Machine learning techniques have obvious applications in this area since they are commonly used for pattern recognition and classification. Two such systems were implemented as a means of demonstrating this capability. Of these a naïve Bayes system provided the best overall performance from a classification accuracy standpoint, but both systems performed well enough to justify further investigation.

Ultimately, the off-vehicle usefulness of declarative agendas is limited. The next chapter, however, will describe the development of an autonomous vehicle control architecture that more fully implements the semantics of declarative AVCL missions and provides for an increased level of autonomy to vehicles currently relying on more primitive control architectures.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. THE EXTENDED RATIONAL BEHAVIOR MODEL (ERBM) DEVELOPMENT AND IMPLEMENTATION

A. INTRODUCTION

In and of itself, the use of declarative missions as described in Chapter VI does not provide a terribly strong case for the usefulness of AVCL's declarative goal-based mission definition functionality. There is simply no way to expose, much less take advantage of, all aspects of a declarative agenda short of an on-vehicle implementation that monitors and adapts mission flow to a developing situation. However, the techniques of Chapter VI for pre-mission conversion of declarative agendas to task-level behavior scripts can be used as the basis for a multi-layer autonomous vehicle control architecture.

There are two advantages to developing an autonomous vehicle control architecture around the capabilities of a data model along the lines of AVCL. First, it allows for more abstract mission definition than the majority of existing architectures. That is, mission definition is independent from the behaviors that will ultimately drive the vehicle. Further, the translation mechanisms described in Chapter V make it possible to install the control architecture on virtually any vehicle with minimal modification to the existing control system. In effect, the nature of the data model itself facilitates the development of a multi-layer control architecture that can be used with virtually any vehicle.

The exemplar control architecture that was implemented for this dissertation is an expanded version of the previously discussed RBM. The basic relationship of this Extended RBM (ERBM) architecture to the AVCL data model and an existing vehicle controller is depicted in Figure 7.1. At the top level, the ERBM Strategic level controls the execution of a declarative agenda by issuing task-level behavior scripts to the Tactical level. The Tactical level is responsible for controlling execution of the most recent script and does so by issuing individual behaviors to the Execution level. Since the Execution level is effectively comprised of the vehicle's existing control architecture, it is necessary to translate behaviors to the target vehicle's native command syntax as they are issued. The only vehicle-specific requirements are provisions for the existing architecture to

receive and execute commands from the ERBM controller, and similarly to provide the Strategic and Tactical levels access to adequate vehicle state information (e.g., telemetry, sensor and system data) to maintain the appropriate level of situational awareness.

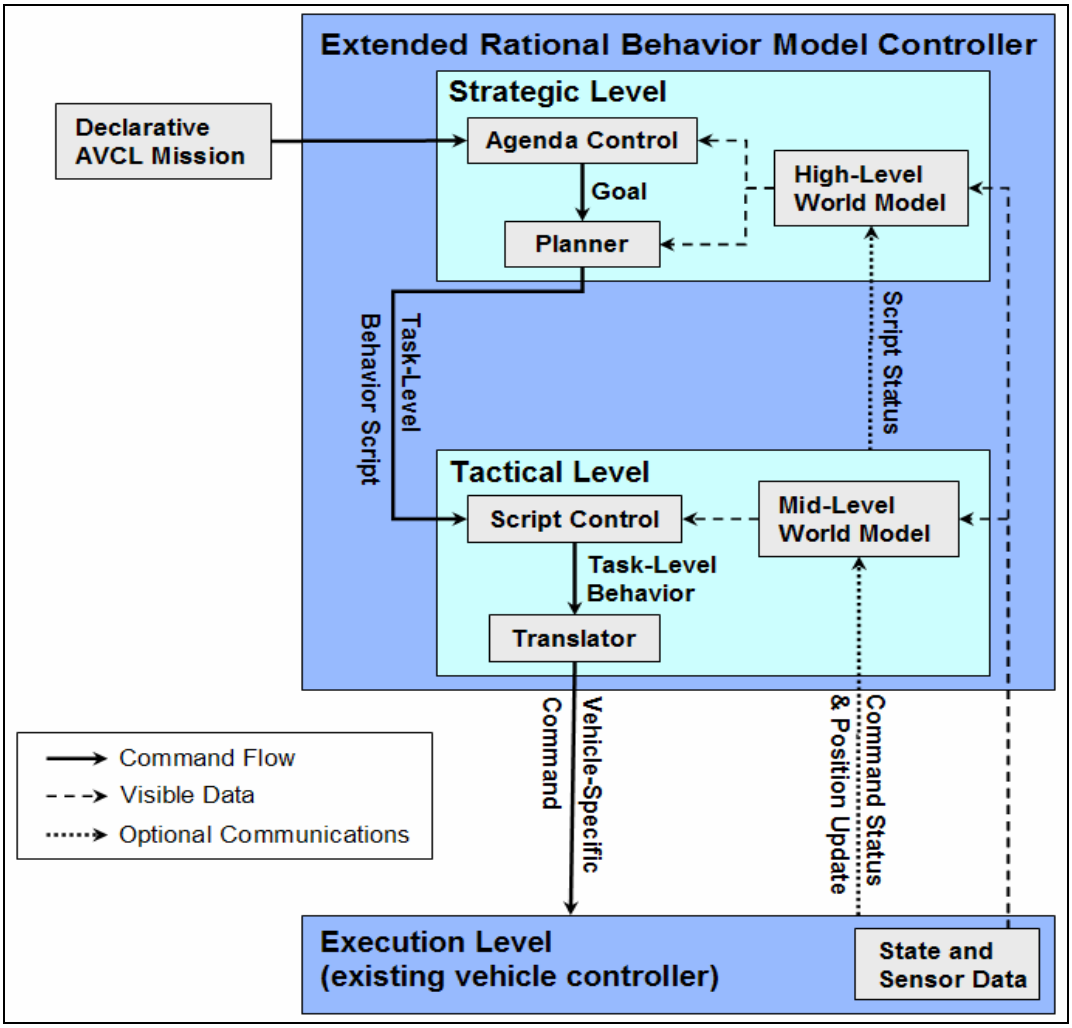


Figure 7.1. The Extended Rational Behavior Model (ERBM) Data and Command Flow for a Typical On-Vehicle Implementation

The remainder of this chapter provides a description of the ERBM, particularly where it differs from the originally proposed RBM, and a discussion of the implementation details on the NPS ARIES UUV.

B. THE EXTENDED RATIONAL BEHAVIOR MODEL (ERBM)

1. Overview

As stated in the RBM overview of Chapter II, the RBM is modeled on the command structure of a naval vessel. The ERBM maintains this conceptualization, but

modifies a number of aspects. Since the Execution level consists of the vehicle's own control system, it is not addressed by the ERBM. However, functionality along the lines of the RBM Execution level is assumed and the conceptual relationship between the ERBM Execution level and the rest of the architecture remains the same. Specifically, it is assumed that the Execution level is responsible for the software to hardware interface and maintains vehicle stability as it executes the behaviors specified by the Tactical level.

In order to facilitate simplicity and determinism, the characteristics, interfaces and even the programming paradigms of the RBM Strategic and Tactical levels are rigorously defined (Byrnes, 93). Table 7.1 lists relevant Strategic and Tactical level restrictions. Many of these characteristics, however, are unrealistic given the level of abstraction of AVCL goals. The ERBM, therefore, modifies or relaxes several characteristics.

RBM Strategic Level
Symbolic computation only. Contains the mission specification and doctrine.
No storage of internal vehicle or world state variables.
Rule-based implementation made up of a rule set and inference engine.
Non-interruptible. Not event driven.
Directs the tactical level through asynchronous message passing.
Messages may be either commands or queries requiring Boolean responses.
Operates in the discrete domain independent of time.
Building block: goals.
RBM Tactical Level
Provides asynchronous interface between Strategic and Execution levels.
Behaviors reside here and may execute concurrently.
Behaviors are implemented as methods of objects.
External interface consists of two parts: behavior activations from the strategic level and command / telemetry paths to / from the Execution level.
World and mission models maintained here.
Responds to Strategic level queries with a logical TRUE / FALSE.
Not interruptible except for data transfers. Hard deadlines cannot be guaranteed.
Operates in the discrete event / continuous time domains.
Building block: programming objects with behaviors.

Table 7.1. Characteristics of the Strategic and Tactical Levels of the RBM as Defined in (After: Byrnes, 93)

Rather than focusing on implementation details of the individual levels, the ERBM concentrates on the level of decision-making required at various levels and the interfaces between levels. Implementation details are not constrained so long as the communication model is adhered to. Determinism is not built into the model itself, but is

dependent on the implementation. It is permissible for different implementations to use different behavior sequences to accomplish the same goal. It is important, therefore, to differentiate between the requirements of the ERBM and the details of a particular implementation.

2. The Strategic Level

The Strategic level of the RBM consists of a set of rules and the inference engine that resolves them. When implemented using a symbolic programming language along the lines of Prolog or the C-Language Integrated Production System, the mission specification and operating system make up the entire Strategic level implementation (Byrnes, 93). This is not possible with AVCL goals since they are not directly executable. The ERBM Strategic level, therefore, must be implemented to load, interpret and execute arbitrary AVCL agendas (i.e., the mission specification is an input to, not a part of, the Strategic level). The role of the ERBM Strategic level is to generate task-level behavior scripts that will accomplish the goals defined in the AVCL agenda. It accomplishes this through the application of techniques along the lines of those described in Chapter VI in response to the real-world events. The characteristics of the ERBM Strategic level most closely related to those of the RBM are listed in Table 7.2.

ERBM Strategic Level
Contains the mission specification.
Symbolic and numerical computation as required.
Implementation must support real-time goal decomposition into task-level behavior scripts.
Maintains a world model sufficient for determination of goal success and failure.
Interruptible, partially event driven.
Directs the tactical level through asynchronous message passing.
Messages may be either task-level behavior scripts or queries.
Operates in the discrete domain with limited dependence on time.
Task building block: AVCL goals.

Table 7.2. Characteristics of the ERBM Strategic Level

Not surprisingly, there are a number of significant differences between the ERBM Strategic level and the Strategic level of the original RBM. First, the ERBM Strategic level definition does not prohibit numerical computation and allows the use of vehicle or world state information. Nevertheless, numerical computation is not elevated as a core Strategic level responsibility. Rather, such non-symbolic operations are permitted in

order to facilitate the implementation of a world model and planning algorithms in support of task-level behavior script generation. Examples might include either periodic sampling of vehicle telemetry to facilitate monitoring of goal execution status, or definition of the operating area in Cartesian space for the purpose of search pattern development. In short, neither the type of computation conducted nor the type of information maintained at the Strategic level is constrained by the ERBM, but the computation and information must still directly support the generation of task-level behavior scripts that accomplish the goals defined by the agenda.

Communication with the Tactical level takes two forms. When the Strategic level develops a course of action in the form of a task-level behavior script, it is immediately sent to the Tactical level. It is expected that the Tactical level begins execution of a script immediately upon receipt even if it requires the interruption of a currently executing script. In this way, the Strategic level can replan as required when new information is received. The second method of communication with the Tactical level is through queries. Whereas the RBM only allowed queries requiring a Boolean response, the ERBM implements a set of query messages that can be used to request various types of status information. The Tactical level is expected to provide the requested information using the appropriate response. Available ERBM inter-level messages are described in Table 7.3.

A final difference between the RBM and ERBM Strategic levels is that the ERBM Strategic level is interruptible and partially dependent on timing. This serves two purposes. The first purpose is to support AVCL goal-timing requirements. Since determination of goal success is a Strategic-level responsibility, it is at this level that a failure must be triggered if a goal fails to succeed in the allotted time or the vehicle does not arrive in the operating area by the designated start time. The second purpose is to support a shift from the RBM communications model, in which lower levels provide information only upon request by the next higher level, to a model where the lower level provides information whenever it becomes available. Under this model, the Tactical level can interrupt the Strategic level and potentially initiate replanning at any time by providing information concerning the status of the currently executing script.

Message	From	To	Description	Parameters
QueryPosition	Strategic	Tactical	Request for the most recent vehicle position.	none
	Tactical	Execution		
QueryScriptComplete	Strategic	Tactical	Request for the status of the current task-level behavior script.	none
QueryCommandComplete	Tactical	Execution	Request for the status of the currently active task-level behaviors.	none
Position	Tactical	Strategic	Report of the vehicle position (Cartesian).	X (double), Y (double)
	Execution	Tactical		
ScriptComplete	Tactical	Strategic	Status report for the current task-level behavior script.	status (boolean)
ScriptFail	Tactical	Strategic	Report that the current script cannot be completed.	none
CommandComplete	Execution	Tactical	Status report for the currently active task-level behaviors.	status (boolean)
CommandFail	Execution	Tactical	Report that the active behaviors cannot be completed.	none
TargetFound	Tactical	Strategic	Report that a search objective has been located.	target type (string)
TargetDestroyed	Tactical	Strategic	Report that an Attack or Demolish objective has been destroyed.	target type (string)
ContaminantDetected	Tactical	Strategic	Report that a contaminant has been detected	contaminant type (string)
ContaminantRemoved	Tactical	Strategic	Report that a detected contaminant has been cleaned.	contaminant type (string)
SignalDetected	Tactical	Strategic	Report that a signal of interest has been detected	frequency (double)
StrategicEnding	Strategic	Tactical	Report that the Strategic-Level thread is terminating	none
TacticalEnding	Tactical	Execution	Report that the Tactical-Level thread is terminating.	none

Table 7.3. Available ERBM Inter-Level Messages

This push-pull communications model significantly reduces inter-level communications requirements. Implementation of the ERBM Strategic level using the RBM's pull-only communications model calls for repeated polling of the Tactical level to

obtain updated position and goal-specific information. This requires two messages per polled data item for each iteration of the Strategic level's think-decide-act loop. Implementation using a push-pull or push-only model potentially reduces message requirements by at least 50 percent by eliminating queries. Additionally, the push-pull communications model more accurately mimics the command relationships and interactions of a manned vessel than the pull-only model. This approach may even facilitate placing the ERBM controller off-line from the vehicle further increasing installation flexibility and minimizing on-vehicle installation requirements.

3. The Tactical Level

The functionality of the ERBM Tactical level differs only slightly from that of the originally proposed RBM and many of the characteristics listed in Table 7.1 are largely unchanged. The characteristics of the ERBM Tactical level (Table 7.4) have evolved from those of the RBM to support the use of AVCL task-level behaviors and the push-pull communications model described above. The responsibilities of the Tactical level are twofold. The first is to direct the activities of the Execution level using the current task-level behavior script. The second is to develop and provide the information required by the Strategic level to monitor the status of the current goal, replan when required, and determine goal success or failure.

ERBM Tactical Level
Provides asynchronous interface between Strategic and Execution levels.
Initiates activation and termination of task-level behaviors in accordance with the AVCL behavior activation and termination criteria.
External interface consists of two parts: behavior-script activations from the Strategic level and command/telemetry data paths to/from the Execution level.
Maintains a world model sufficient to determine the execution status of the current task-level behavior script.
Responds to Strategic level queries with a set of predefined messages. May provide status information using these messages without a Strategic level request.
Directs the Execution level by asynchronous message passing.
Messages may be either individual task-level behaviors or queries.
Interruptible by the Execution level only by command failure notification and data transfers. Can be interrupted by the Strategic level at any time to modify tasking.
Operates in the discrete event / continuous time domains.
Task building block: AVCL task-level behaviors.

Table 7.4. Characteristics of the ERBM Tactical Level

The Tactical level directs the Execution level through the issue of individual task-level behaviors. The behavior activation and termination criteria described in Chapter IV must be observed. In many cases the Tactical level simply forwards the behaviors contained in the current script, although it is permissible for the Tactical level to make modifications as long as the they do not conflict with Strategic level directives. As an example, that Tactical level might insert intermediate waypoints to avoid obstacles while transiting between the global waypoints specified by the Strategic level.

The second responsibility of the ERBM Tactical level is to develop and maintain a world model that supports inter-level reporting requirements. Model contents might include detected targets (type, location, classification and status), signals of interest detected, and the vehicle system status effecting the ability to execute task-level behavior scripts. Thus, as with the original RBM, the Tactical level is the appropriate location in the architecture for the inclusion of software modules pertaining to simultaneous localization and mapping, object and feature detection and classification, and mission system or payload management.

The communication interface between the Tactical and Execution levels is semantically similar to the interface between the Strategic and Tactical levels in that it uses a push-pull model. The Tactical level can issue new task-level behaviors to the Execution level at any time. It is assumed that one or more of the translation mechanisms described in Chapter V will be required to convert the AVCL task-level behaviors to the appropriate vehicle-specific format. Additionally, it is permissible (but not required) for the Tactical level to use the query messages listed in Table 7.3 to request information from the Execution level. The Execution level is expected to respond to Tactical level queries with the appropriate message. It is also permissible for the Execution level to provide information without its being requested.

It is worth noting that the only required modifications to the existing vehicle controller are the implementation of data-passing interfaces with the Tactical level and dynamic behavior script activation. These can be developed in a manner appropriate to the vehicle on which the ERBM controller is to be utilized. Networked message passing, piped inter-process communication and shared memory are all viable options depending

on the implementation circumstances. For the time being, it is assumed that the Tactical level has at least limited access to the vehicle's raw sensor data since messages have not yet been implemented to request or transfer this information. For this and other reasons, a great deal of future work can be applied to the ERBM Tactical level.

In summary, whereas the Strategic level decomposes goals into task-level behavior scripts for issue to the Tactical level, the Tactical level decomposes task-level behavior scripts into individual behaviors for issue to the Execution level at the appropriate time. Additionally, the Tactical level is responsible for interpreting the raw data available at the Execution level in order to identify and report significant events along the lines of target detections and system malfunctions. A push-pull communications model is utilized throughout to best exchange information between levels.

4. Exemplar ERBM Implementation

a. Strategic Level Implementation

Development of an exemplar ERBM implementation in the course of this work focused primarily on the Strategic level. Among the most important choices to be made during implementation of the ERBM Strategic level is the nature of the decision mechanism since it determines when planning and replanning are required, how that planning is conducted, and when a goal has succeeded or failed. In the development of multi-layer vehicle control in support of NPS ARIES rendezvous with another UUV, a finite state machine was chosen because it provides a clear framework for the design of the logic of the rendezvous process and also provides a mechanism for high-level control over the process (Nicholson, 04). Building on the success of this implementation, the ERBM implementation described here also relies on finite-state-machine-based control at the Strategic level. Thus, the Strategic level can accurately be described as nested finite state machines. As with the original RBM, a mission-level state machine determines which goals are executed in what order. A goal-specific finite state machine controls planning and replanning for each goal in the mission.

At the heart of the Strategic level is a mission-flow controller that contains the agenda's mission-level finite state machine as defined by the goal list portion of the AVCL document, as well as the set of avoid areas listed in the constraints portion of the

agenda (including any avoid areas that might be added dynamically as the mission proceeds). The mission-flow controller maintains a reference to the currently executing goal and executes mission-level finite state machine transitions upon goal success or failure. Additionally, the flow controller instantiates a goal-specific planner along the lines of the rendezvous planner of (Nicholson, 04) for each goal and relays information from the Tactical level to the planner associated with the currently executing goal planner.

Goal-specific planners are responsible for all task-level behavior script generation required for the accomplishment of a given goal. Each planner consists of a goal-type-specific finite state machine, state variables suitable for control of state machine transitions, and methods capable of generating task-level behavior scripts appropriate for each state in the finite state machine. As the mission progresses, state transitions result in replanning based on the new state and immediate relay of the generated task-level behavior script to the Tactical level. Upon reaching the finite state machine's terminal state (which can indicate goal success or failure), the flow controller is notified so that the mission-level state machine transition can be executed and planning begun for the next goal in the agenda. The goal-level finite state machines are depicted in Figures 7.2 through 7.9. Not depicted are the state machines corresponding to the Rendezvous goal type since it is documented in detail in (Nicholson, 04) or the Reposition goal type which consists of only Transit and Complete states. For the sake of simplicity, status reporting is not depicted in these figures, but the transmission of any reports that may be specified by an AVCL goal is implicitly included in the transitions.

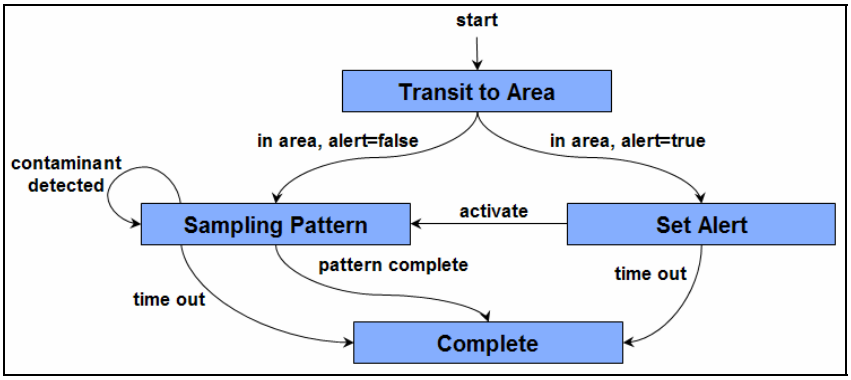


Figure 7.2. A Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL Environmental Sampling Goals

The simplest goal-specific finite state machines (not including the Reposition goal) are associated with the SampleEnvironment, IlluminateArea, Jam, MonitorTransmissions and Patrol goals (Figures 7.2 through 7.5). As with all goal-specific state machines, the start state controls the transit to the operating area. Upon arrival the finite state machine transitions to either an execute or alert state (i.e., be prepared to execute, but do not do so until ordered). The set of task-level behavior sequence for the execute state of a Jam, IlluminateArea or MonitorTransmissions goal directs the vehicle to the center of the operating area and then activates the jammer, illuminator or receiver. The assumptions associated with the inability of the present task-level behavior set to control mission-specific systems are still germane.

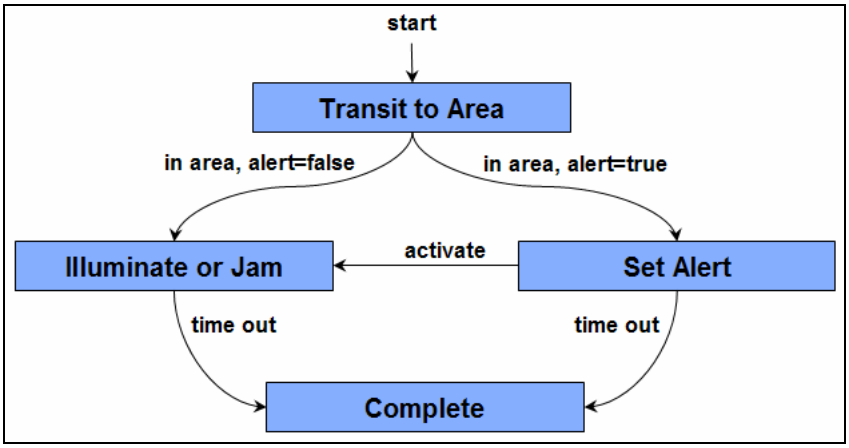


Figure 7.3. A Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL IlluminateArea and Jam Goals

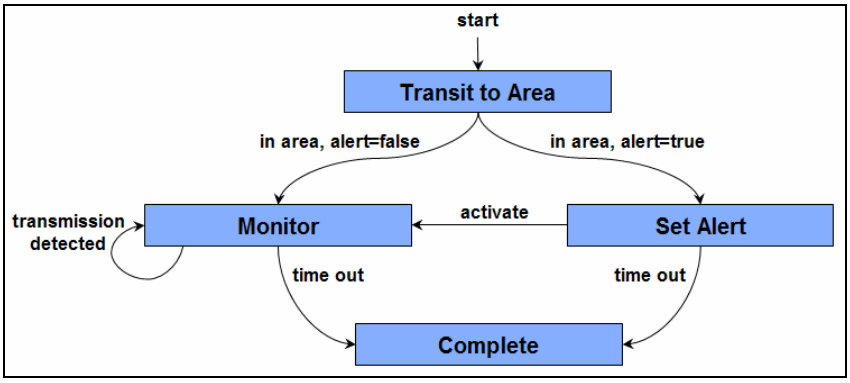


Figure 7.4. A Goal-Type-Specific Finite State Machine for ERBM Strategic Level Use in the Accomplishment of AVCL MonitorTransmissions Goals

The behavior sequence for the execute phase of a Patrol or SampleEnvironment goal consists of a search pattern generated as described in Chapter VI. A transition from the execute state or alert state to the complete state (which can indicate either goal success or failure) is executed at the goal's designated end time. Additionally, the SampleEnvironment state machine transitions to the complete state if the pattern is concluded prior to the designated end time. The Patrol goal state machine, on the other hand, initiates a new search pattern if it finishes the pattern prior to the designated end time. The SampleEnvironment, MonitorTransmissions, and Patrol state machines transition from the execute state back to the same state upon detections to facilitate reporting.

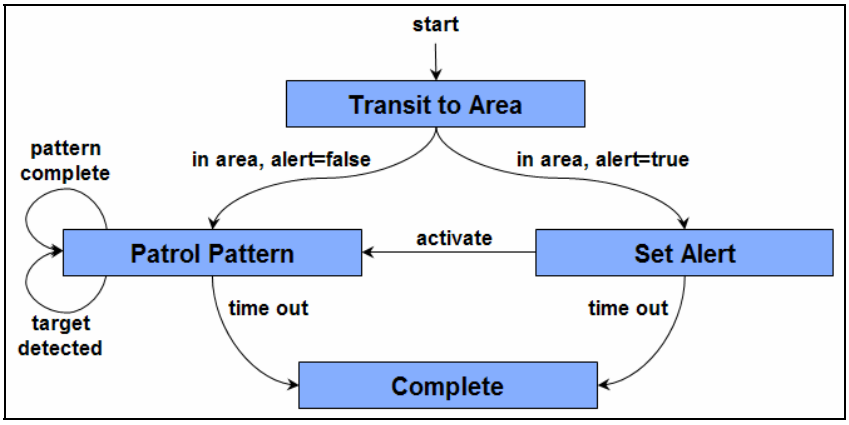


Figure 7.5. A Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL Patrol Goals

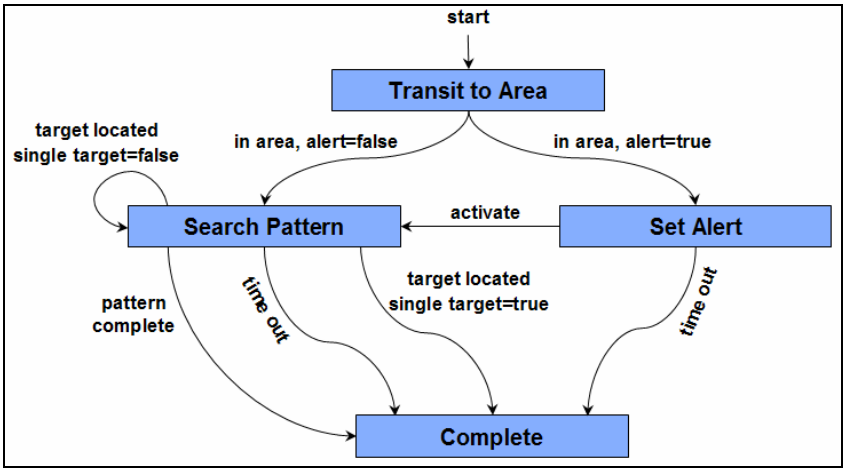


Figure 7.6. A Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL Search Goals

Only slightly more complicated is the finite state machine associated with AVCL Search goals (Figure 7.6). This finite state machine is similar to the one associated with Patrol goals, however it does differ somewhat in the transitions. Since a Search goal can direct the vehicle to search for single or multiple targets, there are two possible transitions from the execute state following a detection. If there is only a single search objective, a transition to the complete state is appropriate. If there are potentially multiple targets, the post-detection transition is back to the execute state so that the search can continue. Upon pattern completion, the finite state machine transitions to the complete state regardless of the timing since the search pattern was specifically planned to achieve the probability of detection ordered by the goal.

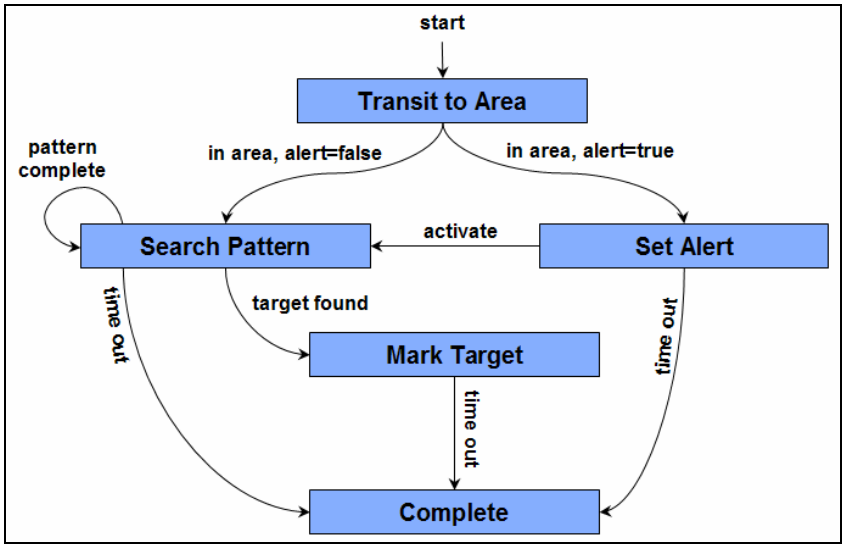


Figure 7.7. Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL MarkTarget Goals

The goal-specific finite state machines associated with the remaining goal types (MarkTarget, Decontaminate, Attack and Demolish) each build upon one of the preceding state machines. The MarkTarget finite state machine (Figure 7.7) is similar to the Patrol state machine except that upon target detection, the transition is to a mark target state (where it remains until the goal's specified end time) rather than back to the execute state.

Similarly, the Decontaminate finite state machine (Figure 7.8) extends the state machine associated with SampleEnvironment goals. In this finite state machine,

when a contaminant is detected, a transition is made to a decontaminate state. When the contaminant has been successfully removed, the state machine transitions back to the execute state and the sampling (search) pattern continues.

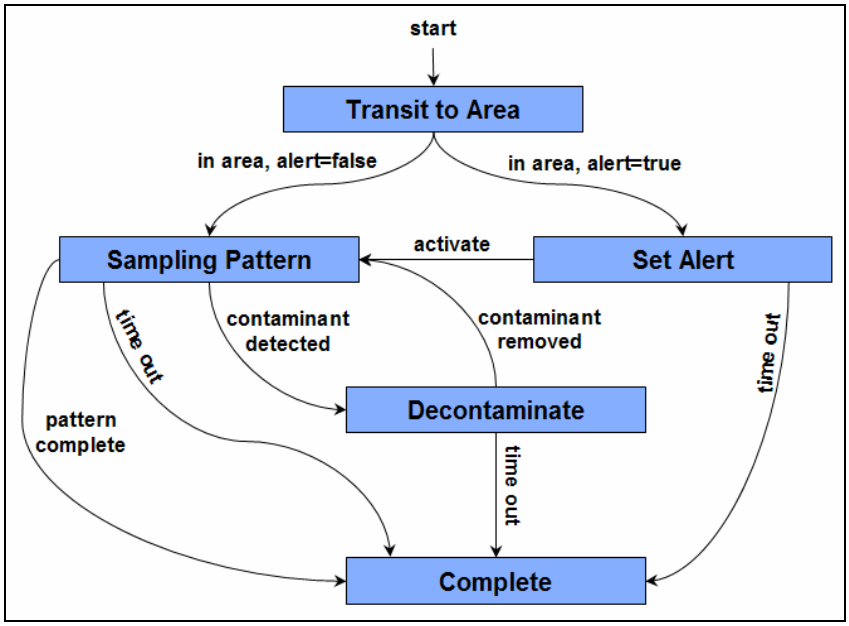


Figure 7.8. Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL Decontaminate Goals

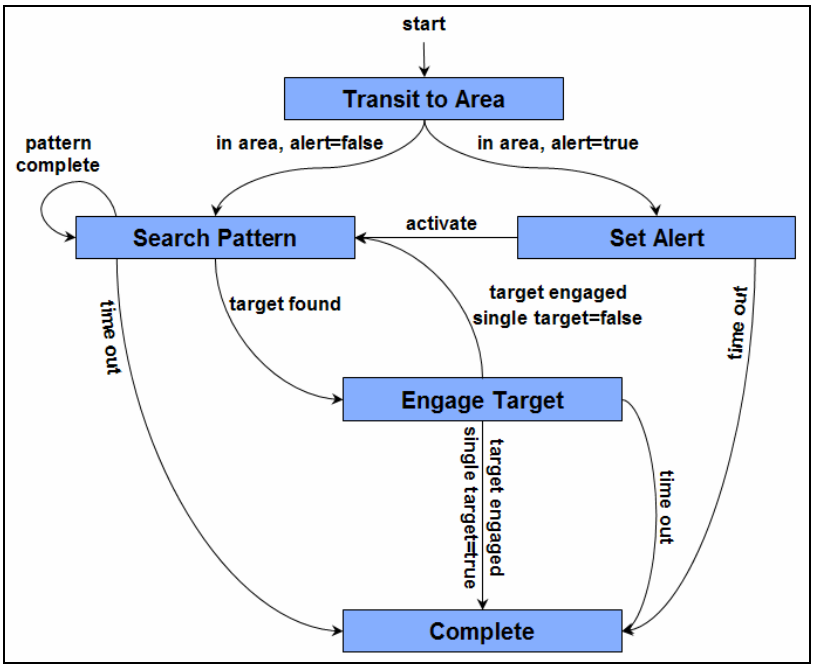


Figure 7.9. Goal-Type-Specific Finite State Machine for ERBM Strategic Level use in the Accomplishment of AVCL Attack and Demolish Goals

Finally, the finite state machine used to control the execution of Attack and Demolish goals (Figure 7.9) is an extension of the one associated with Patrol goals. The added state here is an engage state that controls the actual attack or demolition of the encountered target. The transition from this state is executed when the target has been successfully engaged with the controller returning to the execute state if the goal calls for the attack of multiple targets or transitioning to the complete state the goal calls for the attack of only one. Regardless of the current state, the controller transitions to the complete state if the goal times out.

It must be noted that while the finite state machines associated with the goals imply full functionality, scripts cannot presently be generated to fulfill the requirements of the added state (mark target, decontaminate, or engage). As with pre-mission use of these goal types, full implementation in the ERBM can likely be facilitated by the development of a standardized mission system interface functionality that is incorporated into the task-level behavior set.

b. Tactical Level Implementation

At the Tactical level, the current exemplar implementation issues individual task-level behaviors from the current script to the Execution level at the appropriate times. Thus, it complies with the ERBM Tactical level requirements discussed in the previous section, but at the same time it has significant room for growth. Interpretation of sensor data, onboard systems monitoring and control, local path planning and obstacle avoidance, and object classification in particular are areas in which Tactical-level functionality can improve. However, given the numerous ongoing research efforts in these and other areas that might be applied at the Tactical level, the decision was made to focus ERBM development on other aspects of the architecture for the time being. Ultimately, improvement of the Tactical level functionality through the addition of capabilities such as obstacle avoidance, contact detection and classification, and simultaneous localization and mapping provides a number of potential areas for future work relating to the proposed ERBM.

C. RBM STRATEGIC AND TACTICAL LEVEL IMPLEMENTATION ON THE ARIES UUV

1. The Existing ARIES Control Architecture

The internal configuration of the NPS ARIES vehicle upon which the ERBM controller exemplar is installed is depicted in Figure 7.10. Onboard computers include a PC/104 stack containing two Pentium III processor boards, two 40 gigabyte hard drives and various input / output cards. The two processors operate as independent computers, designated QNXE and QNXT, each running the QNX Neutrino real-time operating system (QNX, 05). Although independent, the QNXE and QNXT computers maintain a shared memory block that allows processes on either computer to share information without the overhead of network or inter-process communication. The QNXE computer controls mission execution and directly interacts with the vehicle control actuators. The execution process (RExec) runs on this computer and controls the vehicle as directed by the track.out waypoint file. The QNXT computer, on the other hand, is primarily responsible for navigation and sensor processing and provides filtered navigation and sensor data to the QNXE RExec process and other onboard systems.

A second PC/104 stack containing a single Pentium III processor board and an 80 gigabyte hard drive is available for non-real-time processing. This computer, designated PC104, runs the Windows XP operating system and is used for non-time-critical processing. In the existing vehicle configuration, the PC104 computer is used primarily for sonar and video image processing and interpretation. It communicates with the QNXE and QNXT computers via an onboard 10Base2 Ethernet connection. It is on this computer that the ERBM controller is installed.

The normal means of communication between off-board systems and onboard computers is via wireless network connection. Limited communication with ARIES while the vehicle is submerged is provided by a Benthos acoustic modem. This connection is used to monitor vehicle location during a mission and direct vehicle activity over the course of a mission as described in (Marr, 03). Additionally, the acoustic modem is used for inter-vehicle communication with other submerged vehicles and network nodes.

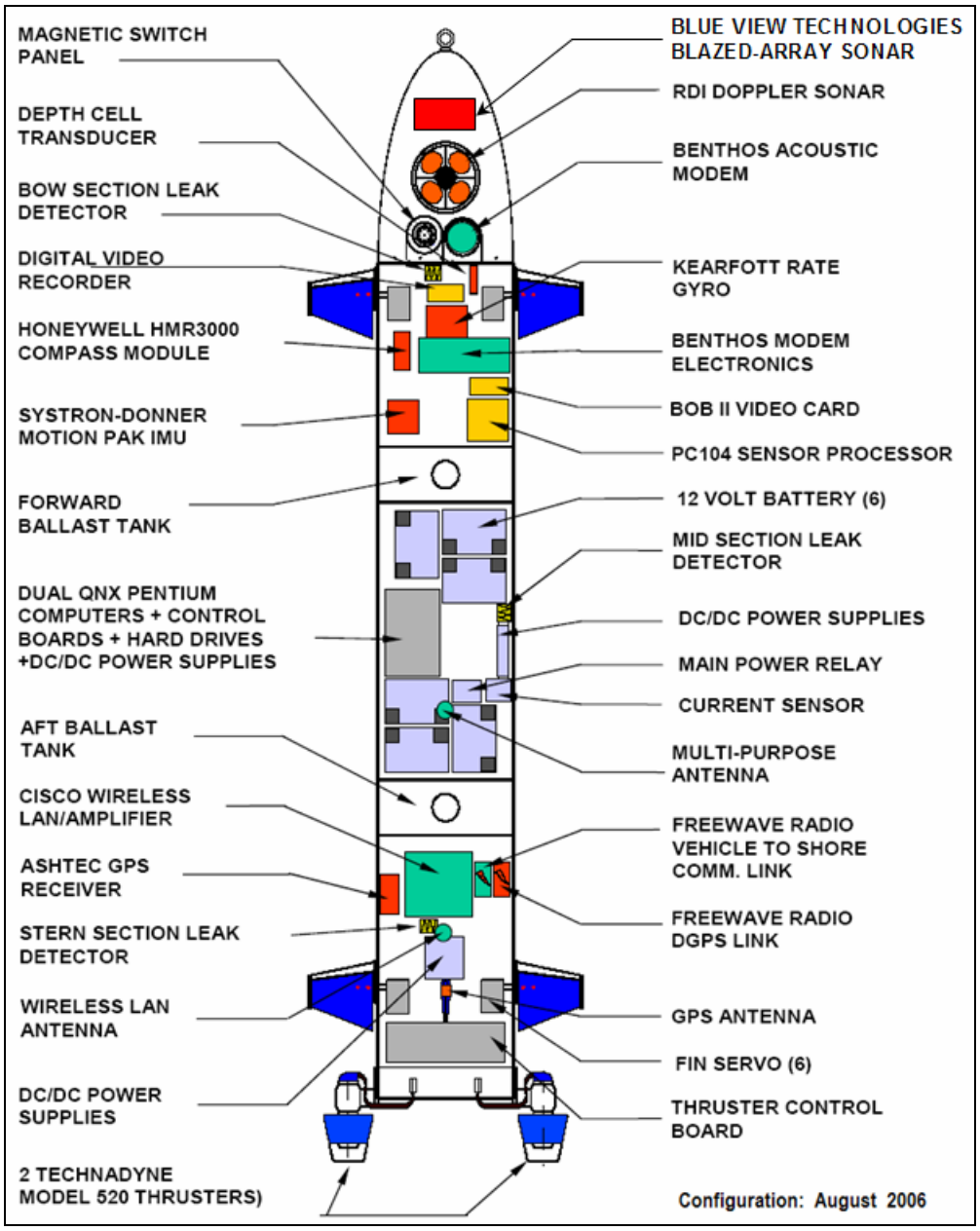


Figure 7.10. The NPS ARIES UUV Configuration (After: Marco, 01)

2. Incorporation of ERBM onto the Existing ARIES Control Architecture

The ERBM controller is implemented in the Java programming language and installed on the ARIES PC104 computer. A single multi-threaded process implements the ERBM Tactical and Strategic levels as well as all required translators and network connections. Future improvements to the Tactical level (i.e., simultaneous localization and mapping, target and obstacle detection and classification, etc.) might be implemented

as threads in the existing Java program or as separate processes (written using any programming language) that communicate with the ERBM controller's virtual machine via a network connection.

At run time, the ERBM controller is started prior to the QNXE RExec process. The controller loads the AVCL declarative mission, initializes the mission-flow controller, and establishes a server socket for the QNXE processes to connect to as required. The ERBM controller accepts multiple connections and each connection is configured as read-only (to the PC104), write-only (from the PC104), or read-write. Beyond this distinction, all connections are treated identically. All read-only and read-write connections can send any valid message to the ERBM controller at any time. Similarly, any message or translated task-level behavior to be sent from the ERBM controller will be transmitted over all active read-write and write-only connections. Strategic-level goal and mission timing commences when the first connection (regardless of type) is established. The current ARIES ERBM implementation uses read-only and write-only connections exclusively.

All modifications and additions to existing ARIES control software are implemented on the QNXE computer. The software architecture of the ERBM-controlled execution level is depicted in Figure 7.11. The most significant modification to the existing architecture is the implementation of an ERBMConnection process that waits for new waypoint lists (track.out files) to be issued by the Tactical level of the ERBM controller. Upon receiving a new waypoint list, the ERBMConnection process archives the current track.out file, saves the new one to the hard drive, and sets a shared-memory script-status flag to indicate to the RExec process that a new script is ready to be loaded and executed. When the ERBMConnection process receives a "TacticalEnding" message instead of a track.out file (indicating that from the standpoint of the ERBM controller the mission is complete), it sets the shared-memory flag to indicate that the mission is complete and that the RExec process is to initiate any mission-termination procedures upon achieving the last waypoint in the currently executing script. The ERBMConnection process utilizes a single write-only (from the perspective of the PC104) ERBM connection.

Only minimal modification of the existing RExec process is required to implement ERBM control. Two initialization steps, launching the ERBMConnection process and establishing a read-only connection (from the perspective of the PC104) with the ERBM controller, and two mission-completion steps, terminating the ERBMConnection process and closing the ERBM-controller connection are implemented in the RExec process.

The most significant changes to the RExec process deal with waypoint list execution. At the start of each iteration of the RExec control loop, the shared-memory script-status flag is checked. If the flag indicates that a new track.out file is present, the new file is loaded for execution and the shared-memory flag is reset. If the flag indicates that no new track.out file has been received, execution of the current file continues uninterrupted. If the flag indicates that the ERBM controller has terminated, execution of the current waypoint sequence continues uninterrupted and mission-termination procedures are initiated when the last waypoint is reached. In this case, the shared-memory flag is not rechecked in future closed-loop iterations. Thus, the ERBM controller is able to interrupt and replace the currently executing waypoint list at any time. Ultimately, the ERBM controller determines that a particular waypoint sequence is to be the mission's last whether it is successfully completed or not.

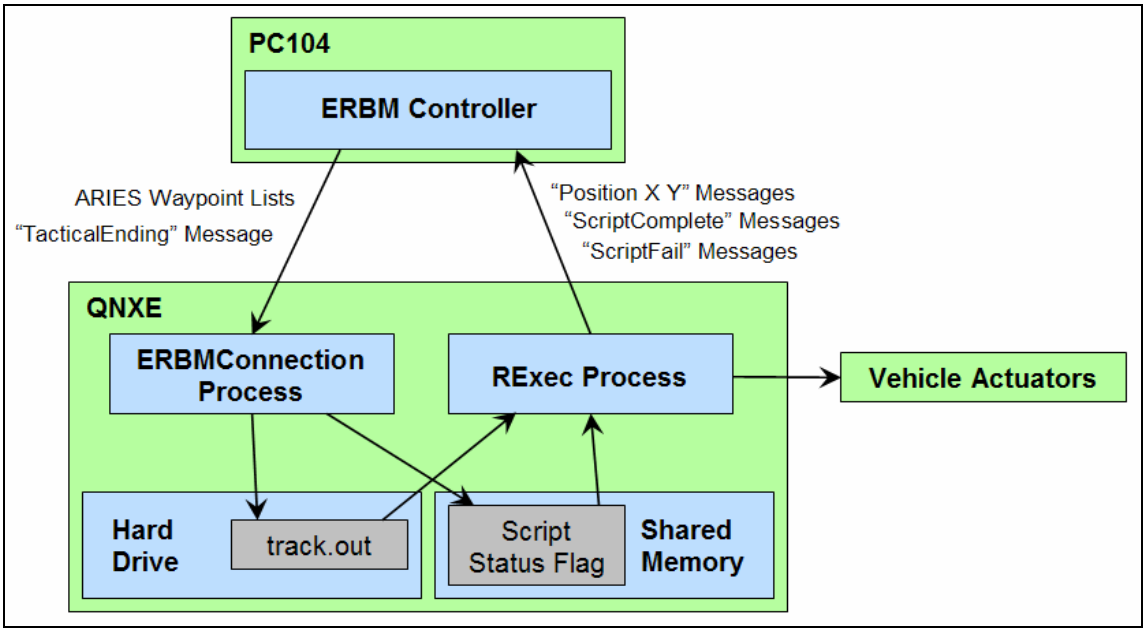


Figure 7.11. The ERBM Controller Implementation on the ARIES UUV

In most cases, the currently executing waypoint list runs to completion without interruption. In these cases, the RExec process must send a "ScriptComplete: true" message to the ERBM controller (unless a "TacticalEnding" message has been received) and continue towards the current waypoint until the controller issues the next waypoint sequence. This requires modification of the existing RExec mission termination criteria since the baseline RExec implementation commences mission completion procedures immediately after reaching the final waypoint of the list. To support ERBM implementation, a three-second delay is instituted to allow time for the ERBM controller to issue the next track.out file. If the ERBM controller does not provide a new script within this time limit, mission termination procedures are initiated without regard to the status of the AVCL mission.

A number of safeguards are implemented to ensure safe vehicle operation. The first is the retention of all existing RExec mission-abort logic. Criteria such as waypoint time-out, leak detection, and hardware malfunctions initiate mission abort procedures regardless of ERBM status, as will an abort order received via the Benthos acoustic modem. In addition, a three-second time-out is implemented both at mission start and following waypoint-list completion to ensure the vehicle does not operate for an extended period of time without a valid control script. Finally, signal handlers are implemented to prevent network problems (i.e., socket errors) from causing the RExec process to terminate unnaturally and leave vehicle actuators, most importantly propellers, active but uncontrolled. Receipt of a socket-related error signal (i.e., SIGPIPE or EPIPE), most likely resulting from unannounced or accidental closure of the socket, terminates networked message transfer between QNXE processes and the ERBM controller. In these cases, the mission is allowed to continue and terminate normally when the final waypoint in the most recently activated track.out file is achieved.

Testing of this ERBM implementation in both real-world and virtual environments is documented in the next chapter as is the connection of the ARIES control software to a six-degree-of-freedom physically-based model in support of simulation testing. It is important to note that this ERBM implementation does not preclude the execution of ARIES missions without ERBM control. In fact, ARIES

waypoint lists can be loaded and run directly (i.e., without ERBM controller use) in exactly the manner as they were prior to ERBM implementation.

D. SUMMARY

The ERBM is a multi-layer hybrid control architecture based on the relationship between AVCL's declarative and scripted mission-definition capabilities. An enhancement of the original RBM architecture, the ERBM improves the capabilities and robustness of the Strategic and Tactical levels while maintaining RBM's correlation to the command hierarchy of a manned vessel.

At the Strategic level, the ERBM applies the techniques described in Chapter VI to convert declarative goals into task-level behavior scripts that are to be executed by the Tactical level. Since these techniques are applied at run time and can generate new scripts at any time in response to real world exigencies, the ERBM is better able to fully utilize all AVCL declarative agenda aspects. The Tactical level controls the execution of task-level behavior scripts generated by the Strategic level by issuing individual behaviors to the Execution level.

The most important improvement over existing autonomous vehicle control paradigms that is provided by the design of a control architecture around the functionality of a data model along the lines of AVCL is its vehicle-independence and the ease with which it can be used to augment an existing vehicle controller. Specifically, the ability to convert AVCL task-level behavior sequences into vehicle-specific tasking allows the ERBM architecture to be installed on top of arbitrary vehicles with minimal modification to the existing vehicle control software (and no modification of the vehicle-specific command set). This A significant increase in the ability to operate in uncertain environments is provided by the realtime planning capability of the ERBM architecture described here.

The ability to implement the ERBM on top of an existing vehicle control architecture is demonstrated by installation on the NPS ARIES UUV. Fairly simple modifications to the existing ARIES control software enables the vehicle to dynamically load and execute waypoint sequences received through a network connection as the mission progresses. The ERBM controller uses the AVCL-to-ARIES XSLT stylesheet to

convert task-level behaviors generated by the Tactical level into ARIES waypoint lists. Results of the ARIES ERBM implementation are discussed in Chapter VIII.

It is also reasonable to infer from the experimental results documented in the next chapter that, the planning algorithms and finite-state-machine-based control used by the ERBM implementation described here provide significantly more robust autonomy than many architectures currently in use. This is not to say, however, that the ERBM architecture is inherently superior to other available autonomous vehicle controllers. Other architectures such as Draper Laboratory's ADEPT and the Pennsylvania State University Advanced Research Laboratory's Intelligent Controller also provide for significant autonomy. In fact, many of the planning algorithms developed in the course of this work might be applicable in systems along the lines of ADEPT and the Intelligent Controller and those implemented within these architectures might prove useful in an ERBM implementation as well.

The primary strength of the ERBM controller is the level of vehicle independence that it achieves and the data-model-based mechanisms by which it achieves this independence. These same mechanisms might ultimately prove useful not only in the evolution of the ERBM architecture, but in the development of other vehicle-independent architectures and the extension of existing architectures as well.

VIII. EXPERIMENTATION

A. INTRODUCTION

This chapter provides a discussion of experiments conducted in support of this work. Experiments are designed to test the procedures and assertions introduced in previous chapters. The preponderance of both simulated and in-vehicle experimentation involves the development and implementation of the ERBM. Thus, ERBM testing comprises the bulk of this discussion. Conversions between AVCL and vehicle-specific data formats are discussed in great detail in Chapter V and receive only brief attention here. The successful implementation of the ERBM controller on the ARIES UUV, however, relies heavily on the stylesheet-based AVCL-to-ARIES conversion, so the ability to automatically convert AVCL to vehicle-specific formats is implicitly demonstrated. Similarly, the techniques for generating task-level behavior scripts to accomplish declarative goals (Chapter VI) is demonstrated by the ARIES ERBM implementation and the simulation results for other vehicle types. Finally, testing of the case-based reasoning and naïve Bayes systems for inferring declarative goals from task-level behavior scripts is discussed in Chapter VI rather than here.

This chapter begins with a description of mission simulation in the AUVW. This is immediately followed by a description of the physically-based models used by the AUVW during simulations. Section C provides a brief discussion of translations between AVCL and vehicle-specific formats. Also provided in Section C is a discussion of experimental results of declarative agenda missions run using the ERBM controller. ERBM results are provided for simulated UUV, USV, and UAV missions as well as real-world UUV (ARIES) missions.

B. MISSION SIMULATION

1. Overview

Autonomous Vehicle missions are simulated in a virtual environment in support of this work using the AUVW. Described in more detail in Appendix B, the AUVW is Java application for mission planning and rehearsal and includes utilities for conversion between AVCL and vehicle-specific tasking languages as described in Chapter V. Additionally, the AUVW incorporates physically-based UUV, USV, and UAV models

for use in simulation and three-dimensional mission visualization using interactive X3D scenes and the IEEE Distributed Interactive Simulation (DIS) protocol (IEEE, 95).

There are two methods of running simulated missions using the AUVW. The most common relies on vehicle-type-specific execution software incorporated directly into the AUVW. Each AUVW vehicle-type-specific component implements the full set of applicable task-level behaviors. Additionally, the ERBM controller is available to provide high-level control of mission execution during declarative agenda simulation. This mode of mission simulation is useful for testing mission flow and vehicle performance during the execution of missions defined with AVCL.

The second method of simulating mission execution using the AUVW models involves their use as a simulation engine for in-vehicle control software. When run in this mode, the appropriate vehicle-type-specific model is executed by itself and establishes a server socket for on-vehicle control software connection. After establishing a connection with the AUVW model, the on-vehicle software runs normally except that it uses simulation values instead of onboard sensor-derived values. At the point in the closed-loop control cycle responsible for reading or computing telemetry and sensor values, a string containing a white-space-delimited series of current telemetry values and control settings is transmitted to the AUVW model. The model uses the telemetry and control information to calculate updated telemetry and sensor information that are transmitted back to the vehicle software using a similar telemetry string. Common telemetry-string fields used by all AUVW models are listed in Table 8.1. Additionally, each telemetry string contains 50 additional values that are used to transmit vehicle-type-specific control and sensor settings and values. The interpretation of each of these model-specific values is available in the AUVW documentation.

All simulation results documented in this chapter for USV and UAV missions were obtained using embedded AUVW vehicle-execution components. UUV simulation results, on the other hand, were obtained using NPS ARIES UUV on-vehicle software communicating with the AUVW model using a network. All simulations were run against the appropriate vehicle-type-specific model. The remainder of this section provides a description of each of the vehicle-type-specific models.

Telemetry String Field	State Variable	Description
1	flag	Either "uuv_state", "usv_state" or "uav_state".
2	t	Current vehicle-execution time (seconds since start).
3	x	X location (meters) of the vehicle in the earth-fixed coordinate frame.
4	y	Y location (meters) of the vehicle in the earth-fixed coordinate frame.
5	z	Z location (meters) of the vehicle in the earth-fixed coordinate frame.
6	φ	Bank Euler angle (degrees). Rotation about the X axis.
7	θ	Pitch Euler angle (degrees). Rotation about the Y axis.
8	ψ	Yaw Euler angle (degrees). Rotation about the Z axis.
9	u	Linear velocity (meters per second) along the body-fixed coordinate frame X axis.
10	v	Linear velocity (meters per second) along the body-fixed coordinate frame Y axis.
11	w	Linear velocity (meters per second) along the body-fixed coordinate frame Z axis.
12	p	Angular velocity (degrees per second) about the body-fixed coordinate frame X axis.
13	q	Angular velocity (degrees per second) about the body-fixed coordinate frame Y axis.
14	r	Angular velocity (degrees per second) about the body-fixed coordinate frame Z axis.

Table 8.1. Autonomous and Unmanned Vehicle Workbench (AUVW) Physically-Based Model Telemetry String Fields Common to all Vehicle Types

2. Physically-Based AUVW Models

Vehicle-type-specific models in the AUVW rely on rigid-body dynamics, Newton-Euler equations, and numerical integration (McGhee, et al., 00). UUV and UAV models are rigorously defined and allow for accurate six-degree-of-freedom modeling of vehicle response. The USV model, on the other hand, provides only two-degree-of-freedom response, making a more accurate USV model, as well as a UGV model, candidates for future AUVW improvements.

A variation of the model described in (Brutzman, 94) is used for UUV modeling in the AUVW. The equations of motion and coefficients of the model correlate directly to the characteristics of the vehicle body and control effectors. The model's relationship between propeller revolutions per minute and forward speed, however, is modified to more accurately reflect actual vehicle response. Rather than the linear speed-per-

revolutions-per-minute coefficient of the original model, three reference speeds (corresponding to 0, 50, and 100 percent of available revolutions per minute) are used to derive a quadratic curve that correlates any revolutions per minute value between 0 and 100 percent to a specific forward speed through the water. All other equations of motion of the UUV model are identical to those documented in (Brutzman, 94).

The model's coefficient values are adjusted as required to accurately model various vehicles. Values used during the tests documented in this chapter model the response of the ARIES UUV and are identical to those of (Brutzman, 94) with two notable exceptions. The ARIES propeller revolutions-per-minute-to-forward speed relationship is based on speeds of 0.0, 1.6, and 1.7 meters per second for 0, 50, and 100 percent of available revolutions per minute respectively. Additionally, since the ARIES UUV does not utilize cross-body thrusters, the values of all coefficients relating to their influence on vehicle response are set to zero.

Providing for only a two-degree-of-freedom response (surge and yaw), the USV model is the most rudimentary of the vehicle-type-specific models currently implemented in the AUVW. Although the model's simplicity effectively precludes its use in testing vehicle response, it does provide a useful tool for evaluating overall mission flow and the progress of a declarative agenda's goals, and is therefore suitable for the types of experiments required to validate the functionality of the ERBM controller.

The model consists of Equations 8.1 and 8.2 where u_{max} is the vehicle's maximum forward speed, r_{max} is the vehicle's maximum turn rate, rpm is the current propeller revolutions per minute setting (or average for multi-propeller vehicles), rpm_{max} is the maximum commandable propeller revolutions per minute, δ_{rudder} is the current rudder deflection (degrees), and $\delta_{rudderMax}$ is the maximum allowable rudder deflection. These equations are used to compute linear acceleration along the vehicle's body-fixed X axis and angular acceleration about the body-fixed Z axis respectively. Results documented in this chapter use coefficients intended to represent a typical medium-speed USV. Maximum speed (u_{max}) was 15.0 meters per second, maximum turn rate (r_{max}) was 0.3 radians per second, maximum revolutions per minute (rpm_{max}) was 1000 and maximum

rudder deflection ($\delta_{rudderMax}$) was 30 degrees. As with other AUVW models, values can be adjusted as required to approximate the response of various vehicles.

$$\dot{u} = 0.2 \cdot \left(\frac{rpm \cdot u_{max}}{rpm_{max}} - u \right) \quad \text{(Eq. 8.1)}$$

$$\dot{r} = 0.33 \cdot \left(\frac{-u|u|\delta_{rudder}}{uu_{max}\delta_{rudderMax}} - r \right) \quad \text{(Eq. 8.2)}$$

The UAV model is based on aerodynamic stability derivatives (Stevens and Lewis, 03) and the equations of motion defined in (Cooke, et al., 92). Coefficient values and descriptions are provided in Tables 8.2 through 8.5. As with other AUVW models, UAV coefficients can be manipulated to model the response of other vehicles as required.

Coefficient	Value	Description
S	12.985	Wing planform area (square meters).
b	14.84	Wingspan (meters).
c	0.875	Average wing chord width (meters).
ε	(-4.5, 0.0, 0.0)	Tail position in the body-fixed coordinate frame (meters).
m	775	Vehicle mass (kilograms).
I_{xx}	9819.59	Inertia tensor xx element.
I_{yy}	7076.06	Inertia tensor yy element.
I_{zz}	16627.7	Inertia tensor zz element.
I_{xy}, I_{xz}, I_{yz}	0.0	Inertia tensor xy, xz, and yz elements.

Table 8.2. UAV Physically-Based Model Vehicle Characteristics

Coefficient	Value	Description
C_{L0}	0.3322	Reference lift at 0^o angle of attack.
C_{D0}	1.772e-2	Reference drag at 0^o angle of attack.
$C_{L\alpha}$	7.556	Lift curve slope.
$C_{D\alpha}$	8.372e-2	Drag curve slope.
C_{M0}	6.718e-2	Reference pitch moment at 0^o angle of attack.
$C_{M\alpha}$	-3.6	Pitch moment due to angle of attack.
C_{LQ}	0.0	Lift due to pitch rate.
C_{MQ}	0.0	Pitch moment due to pitch rate
$C_{L\alpha Dot}$	0.3587	Lift due to angle of attack rate.
$C_{M\alpha Dot}$	-3.771	Pitch moment due to angle of attack rate.

Table 8.3. UAV Physically-Based Model Longitudinal Coefficients

Coefficient	Value	Description
$C_{Y\beta}$	-0.4032	Side force due to side slip.
$C_{L\beta}$	-0.15	Dihedral effect.
C_{LP}	-9.1286	Roll damping.
C_{LR}	0.3599	Roll due to yaw rate.
$C_{N\beta}$	1.4556	Weather-cocking stability.
C_{NP}	-0.3815	Rudder adverse yaw.
C_{NR}	-0.8904	Yaw damping.

Table 8.4. UAV Physically-Based Model Lateral Coefficients

Coefficient	Value	Description
$C_{L\delta e}$	1.229e-2	Lift due to elevator or horizontal stabilator.
$C_{D\delta e}$	6.375e-5	Drag due to elevator or horizontal stabilator
$C_{M\delta e}$	-6.361e-2	Pitching moment due to elevator or horizontal stabilator.
$C_{L\delta a}$	5.35e-3	Rolling moment due to aileron.
$C_{N\delta a}$	0.0	Yawing moment due to aileron.
$C_{Y\delta r}$	2.663e-4	Side force due to rudder.
$C_{L\delta r}$	-8.8738e-4	Rolling moment due to rudder.
$C_{N\delta r}$	-1.049e-2	Yawing moment due to rudder.

Table 8.5. UAV Physically-Based Model Control Coefficients

Although no attempt is made to model the aerodynamic characteristics of an actual vehicle with absolute accuracy, the coefficient values approximate the response of a UAV along the lines of the RQ-1 Predator (Figure 8.1). The coefficient values of Tables 8.2 through 8.5 were derived through testing with a second UAV model based on the summed effects of airfoil sections as described in (Bourg, 02). Airfoil characteristics were obtained from the National Advisory Committee for Aeronautics airfoil tables found in (Abbott and Von Doenhoff, 59). Individual airfoils were chosen and composed in such a way as to reflect the approximate shape and characteristics of the wings, body, and control surfaces of the Predator UAV.

At present, the AUVW does not implement a UGV model, although one will be developed and implemented when required. Thus, this work does not directly address the application of the common data model or the ERBM controller to UGVs. It is, however,

assumed that UGV applicability will ultimately prove similar in principle to the UUV, USV, and UAV results discussed here.

Figure 8.1. The RQ-1 Predator UAV

C. EXPERIMENTAL RESULTS

1. AVCL Translations

A number of task-level behavior scripts and vehicle-specific missions were used to verify the ability to translate between data formats using XSLT and context-free grammars. Vehicle availability, however, allowed for in-vehicle testing with the ARIES UUV only. Correctness of the translations to and from the Phoenix UUV tasking language was verified using the AUVW, which is capable of simulating missions defined in either AVCL or the Phoenix command format. The correctness of the translations for REMUS, Seahorse, and JAUS systems was confirmed using the command-format definitions of (Hydroid, 01), (NAVO, 04), and (JAUS, 04-4) respectively.

In general, both translations from AVCL to vehicle-specific formats and the reverse translations work as described in Chapter V. A number of observations bear mentioning, however. The first is that the same AVCL task-level behavior might be implemented differently by different vehicles. The AVCL UUV waypoint behavior of Figure 8.2, for instance, specifies transit speed as a percentage of maximum available power. Translation for the Seahorse UUV yields the command depicted in Figure 8.3

which specifies transit speed in knots. On the other hand, the REMUS command of Figure 8.4 relies on an open-loop revolutions per minute. Similarly, an ARIES waypoint specifies transit speed as an open-loop voltage to the propeller motors. Ultimately, these three vehicles will use three different transit speeds to execute the same waypoint behavior.

```
<Waypoint>
    <XYPosition x="12700" y="6420"/>
    <Altitude value="4"/>
    <SetPower>
        <AllPropellers value="50"/>
    </SetPower>
    <TimeOut value="500"/>
</Waypoint>
```

Figure 8.2. An AVCL UUV Waypoint Behavior

```
Start_Order               : Waypoint_Navigation_Order
Scheduling_Info_Is_Timed  : False
Destination_Latitude      : 36.716664597583815 Degrees
Destination_Longitude     : -121.81779490517175 Degrees
Transit_Mode              : Steer_to_Line
Transit_Altitude          : 4.0 Meters
Transit_Speed_In_Water    : 3.5 Knots
Use_SSS                   : True
```

Figure 8.3. Translation of the AVCL Behavior of Figure 8.2 for the Seahorse UUV

A second observation is that round-trip translation (i.e., from AVCL to a vehicle specific format and back to AVCL) does not yield an identical result to the original document. Translation of the Seahorse command of Figure 8.3, for instance, results in separate AVCL behaviors for speed and altitude. Additionally, the AVCL result uses the latitude / longitude position and speed in knots of the Seahorse command rather than the Cartesian coordinates and power setting of the original behavior. This does not pose a significant issue in this case because the original and post-round-trip translation behavior sequences are identically translated for the Seahorse UUV.

Similarly, since MetaCommand behaviors that do not apply to a given vehicle are ignored during translation, their content can be lost over the course of multiple translations. A REMUS WaitRun objective is easily incorporated into an AVCL task-level behavior script using a MetaCommand behavior. However, if this script is subsequently translated for use with the ARIES UUV, the MetaCommand behavior will be ignored (although a warning might be generated).

```
[Objective]
Type=Navigate
Latitude=36N42.999875855028904'
Longitude=121W49.067694310305114'
Offset direction=0
Offset distance (meters)=0
Offset Y axis (meters)=0
Minimum range (m.)=20.0
Speed=812.5 RPM
Timeout (seconds)=500.0
Track ping interval (seconds)=0
Follow trackline=Yes
Sidescan range=30
Depth control mode=altitude
Altitude=4.0
```

Figure 8.4. Translation of the AVCL Behavior of Figure 8.2 for the REMUS UUV

Similarly, since MetaCommand behaviors that do not apply to a given vehicle are ignored during translation, their content can be lost over the course of multiple translations. A REMUS WaitRun objective is easily incorporated into an AVCL task-level behavior script using a MetaCommand behavior. However, if this script is subsequently translated for use with the ARIES UUV, the MetaCommand behavior will be ignored (although a warning might be generated).

The previous observations highlight a concern when vehicle-specific data is to be converted to AVCL and then to other vehicle-specific formats (a central premise of this work). In these cases, care must be taken to ensure accurate translation. A consistent geographic origin, for instance, is required to exchange position data between the Seahorse and ARIES UUVs to ensure accurate conversion between positions specified

with Cartesian coordinates and those specified with latitude and longitude. Additionally, open-loop orders can be problematic since they can have different meanings to different vehicles. The use of closed-loop AVCL behaviors, therefore, are more appropriate in most circumstances since they are unambiguous.

Though not without potential pitfalls, the successful implementation of automated translations between AVCL, JAUS, and the command formats of the REMUS, Seahorse, and ARIES UUVs using XSLT, context-free grammars, and the mappings described in Chapter V clearly demonstrate the viability of a common autonomous vehicle data model defined in XML to serve as bridge between various vehicles. Additionally, these translations provide strong evidence of the suitability of AVCL's task-level behavior set (including the implicit behavior initiation and termination criteria) for the tasking and control of arbitrary vehicles. These capabilities are further demonstrated by ERBM results described in the next section.

2. ERBM Testing

a. Overview

The ERBM implementation described in Chapter VII was used to control UUV, USV, and UAV missions. USV and UAV missions were conducted in simulation using the execution software and physically-based models of the AUVW. Since the AUVW execution software implements AVCL task-level behaviors directly, translation to a vehicle-specific format was not exercised in these tests. Thus, testing of the ERBM controller in this manner does not provide direct evidence of its usefulness with vehicles that do not directly implement the AVCL task-level behaviors. It does, however, document the ability of the ERBM controller, associated planning algorithms, and task-level behaviors to provide high-level control for various vehicle types.

On the other hand, both simulated and in-water UUV tests were conducted using the existing ARIES control software. In addition to demonstrating the suitability of the ERBM controller for UUVs, the ARIES experiments directly demonstrate the implementation of a common-data-model-based multi-layer control architecture on a non-data-model-compliant vehicle. An important implication of the ARIES experiments, therefore, is that the ERBM controller is potentially applicable to any vehicle for which a translation XSLT stylesheet is developed.

At the time of this writing, neither the AUVW vehicle-control software nor the ARIES UUV implement target classification or event detection that can provide the ERBM controller information to fully exercise the finite state machine control. For this reason, events were artificially generated in order to force the controller to execute state transitions and replan accordingly. Similarly, the ARIES UUV does not possess sensors appropriate for MonitorTransmisssions or SampleEnvironment goals. However, since these goal-types are appropriate for properly equipped ARIES-like vehicles, missions containing these goal types were conducted in the course of this work. In these experiments, the existence of notional sensors of the appropriate types was assumed.

b. USV and UAV ERBM Results

Mission results of a typical ERBM-controlled USV mission in the AUVW simulation are depicted in Figure 8.5. The mission-level state machine (i.e., the declarative agenda) contained three goals. The first goal directed the point-search of a circular area for a single target. Following target location, the vehicle was to proceed north to jam electronic transmissions. If the target was not located, the vehicle was to proceed to the rectangular patrol area. The agenda called for execution of the Jam goal upon success of the Patrol goal or mission completion upon failure. The mission was also to conclude upon success or failure of the Jam goal. Additionally, the declarative agenda defined three avoid areas.

As indicated in Figure 8.5, the ERBM controller initially directed the vehicle to the first goal's search area, bypassing the circular avoid area with six intermediate waypoints approximating a tangential arc. Upon arriving in the operating area, a sector pattern was commenced in accordance with the decision tree of Figure 6.7 (i.e., point datum and small search area relative to the sensor sweep width). The goal was unsuccessful since the search pattern completed without locating the search target, so the vehicle began execution of the Patrol goal. A parallel-track pattern was dictated because of the area's rectangular shape and the implicit Patrol goal requirement for an area-focused pattern. In the depicted mission, the pattern was completed prior to the end of the patrol period, so the ERBM controller planned and initiated a second pattern, which was interrupted prior to completion at the end of the patrol period (goal successful). The ERBM controller then directed the vehicle to the Jam goal operating area (bypassing the

polygonal avoid area). Following successful completion of the Jam goal (correct jammer operation was assumed), the vehicle proceeded to the designated recovery point.

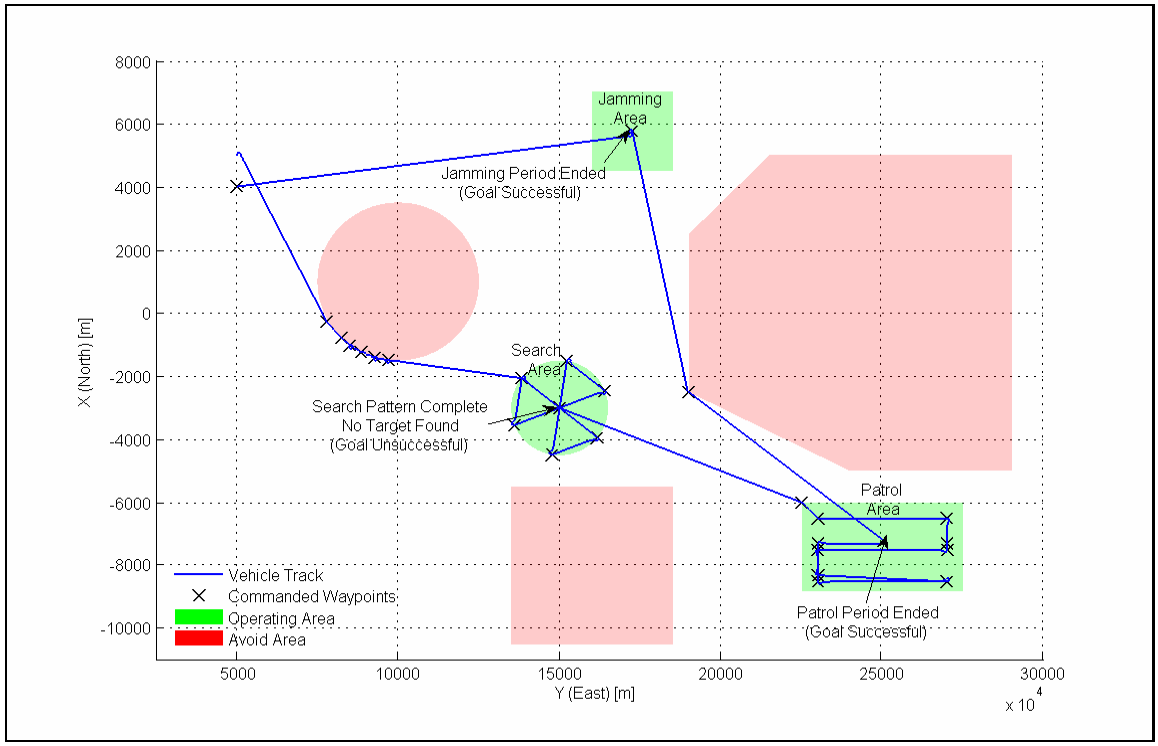


Figure 8.5. Simulated Mission Results for an ERBM-Controlled USV Executing a Declarative AVCL Agenda with Three Goals and Three Avoid Areas

Similar results were obtained for simulated ERBM-controlled UAV missions as indicated by Figure 8.6. The agenda corresponding to the depicted mission summary included a point-focused search, a SampleEnvironment goal, an IlluminateArea goal and three avoid areas. The mission-level finite state machine called for the goals to be executed in order upon success and for the mission to conclude upon the successful completion of the IlluminateArea goal or upon the failure of any of the three goals.

Upon commencing the mission, the ERBM controller directed the vehicle to the first operating area and commenced the expanding square search pattern indicated for a point-focused search of a circular area. After locating the target in the eastern portion of the area, the vehicle proceeded to the SampleEnvironment goal's operating area. The ERBM controller directed a parallel-track pattern appropriate for environmental sampling over a rectangular area and the pattern was completed in the allotted time meeting the criteria for successful goal completion. The vehicle then

proceeded to the IlluminateArea goal's operating area and orbited for the duration of the illumination period before proceeding to the recovery point. As with operation of the jammer in the previous example, correct operation of the illuminator was assumed.

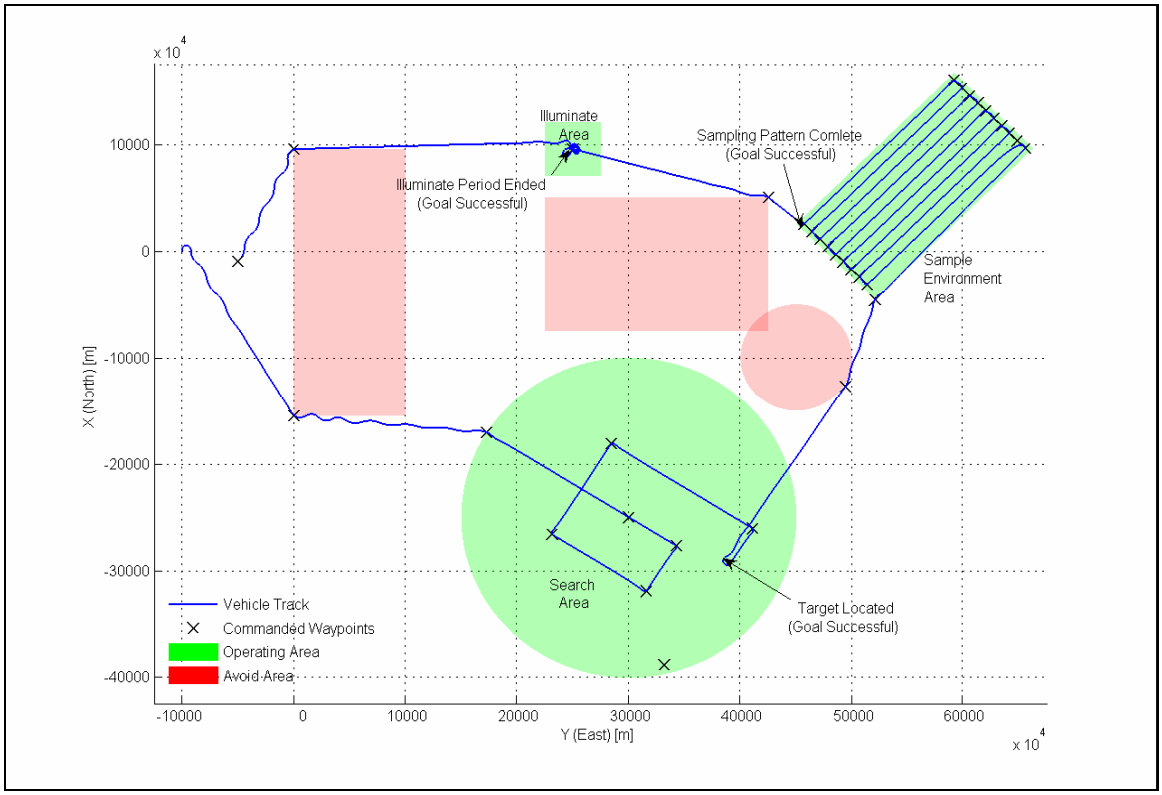


Figure 8.6. Simulated Mission Results for an ERBM-Controlled UAV Executing a Declarative AVCL Agenda with Three Goals and Three Avoid Areas

c. UUV ERBM Results

Because it provides an on-vehicle example of common autonomous vehicle data model application, testing of the ERBM implementation on the ARIES UUV was more comprehensive than with other vehicle types. Initial experiments were conducted to demonstrate ERBM control in the execution of single-goal AVCL agendas with minimal planning requirements. Following the success of these experiments, increasingly complex agendas were attempted, culminating in missions containing multiple goals, multiple avoid areas, and more robust mission-level state machines. When practical, both real-world and virtual environment test missions were conducted. Both simulated and real-world experiments were conducted in real-time using ARIES' existing control software and the ERBM implementation described in Chapter VII.

The first ARIES ERBM test consisted of the simulated and in-water execution of an AVCL agenda with a single Reposition goal and no avoid areas. Possibly the most straight-forward AVCL goal type, the Reposition goal simply calls for the vehicle to transit from its current position to a new location. Optional intermediate points can be included in the goal specification to dictate transit routing. The Reposition goal of the experimental mission included five intermediate waypoints that directed the vehicle first to the north and then to the south prior to proceeding to the recovery point. Summaries of the virtual environment and in-water results are provided in Figures 8.7 and 8.8 respectively. As the figures indicate, the agenda was executed as desired with the vehicle transiting from the launch point to the recovery point while visiting each of the intermediate waypoints. Variations between the simulated and in-water track can be attributed to perturbations of the real-world environment, imperfect in-water navigation, differing vehicle launch headings, and the experimentation with different steering equations. While not requiring deliberative planning or decision-making, this simple experiment does demonstrate the use of automated AVCL task-level-behavior translations as part of the ERBM implementation on a non-AVCL vehicle.

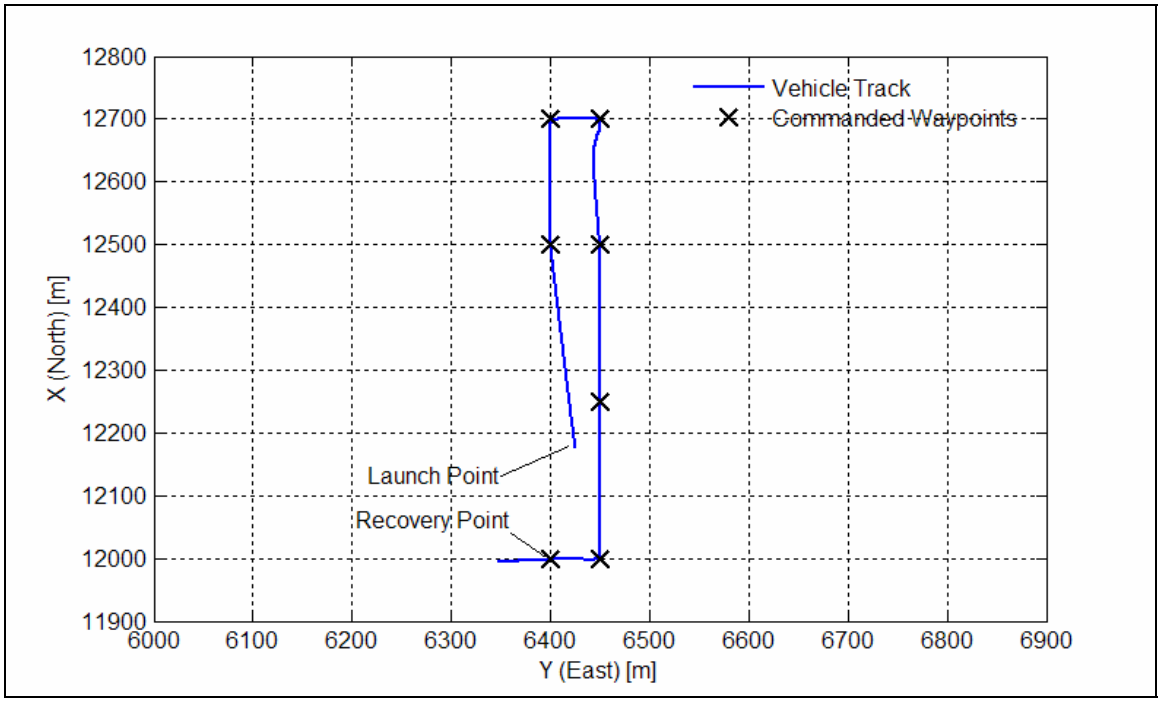


Figure 8.7. ARIES UUV Virtual Environment Results for an ERBM-Controlled Mission with a Single Reposition Goal and No Avoid Areas

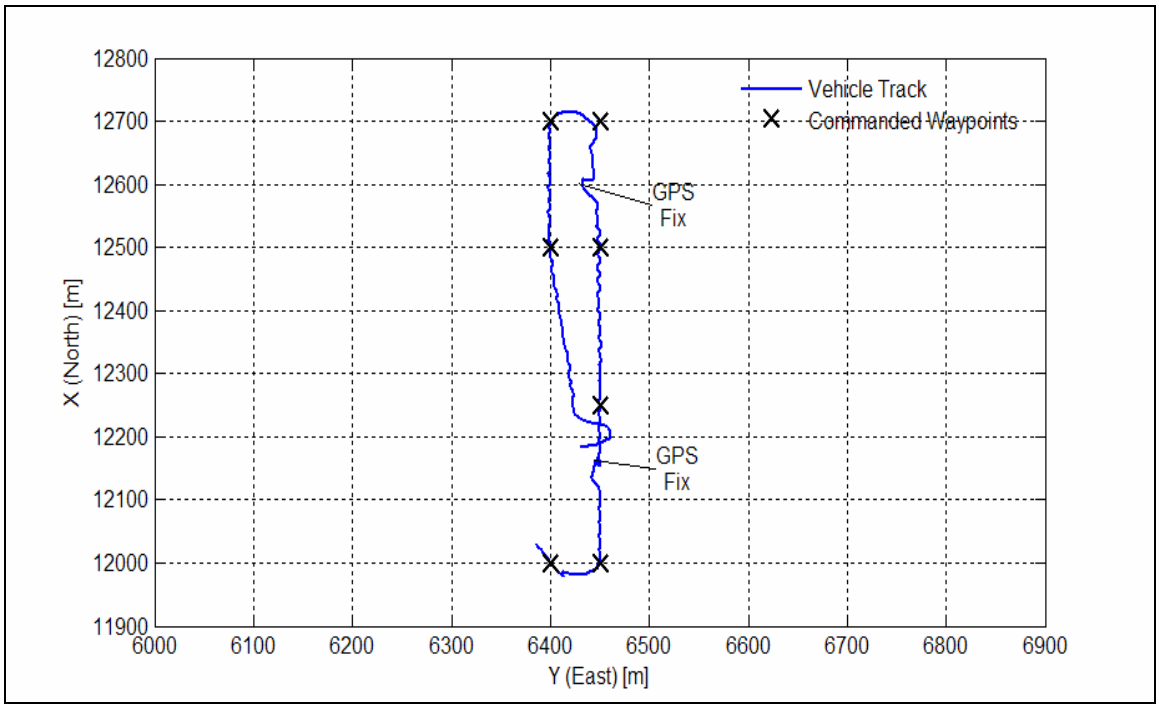


Figure 8.8. ARIES UUV In-Water Results from Monterey Bay (16 June 2006) of the Reposition Mission of Figure 8.7

Extending upon the success of the preceding experiment, a more complex Reposition goal was attempted in the AUVW (no in-water test was attempted because of a desire to focus limited in-water experimentation on more complex agendas). Including multiple avoid areas but specifying only two intermediate transit points, successful execution of this mission required the ERBM path planner to generate, translate, and issue intermediate waypoints to bypass the avoid areas while utilizing the designated routing points. The simulation results depicted in Figure 8.9 indicate that the ERBM controller did exhibit this capability with a total of nine intermediate waypoints being generated (in addition to those specified in the agenda) using the algorithm described in Chapter VI. Thus, the ERBM implementation provides a high-level path-planning capability not inherently available with predefined ARIES waypoint lists. This capability is implicitly exercised in the execution of more complex agendas when transiting to and from the operating areas for various goals.

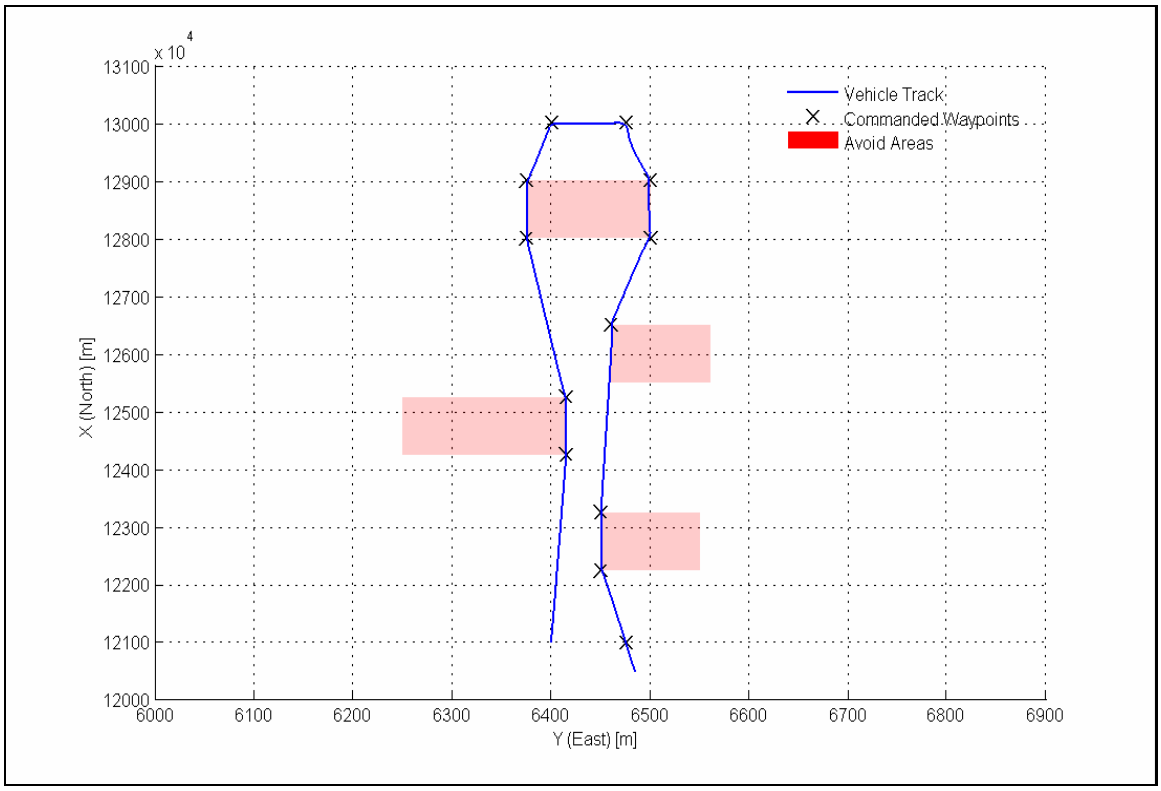


Figure 8.9. ARIES UUV Virtual Environment Results for an ERBM-Controlled Mission with a Single Reposition Goal and Multiple Avoid Areas

Although still not calling for significant goal-achievement planning, a MonitorTransmissions goal does call for more robust implementation than a Reposition goal because it ultimately relies on a Loiter behavior to maintain position in the center of the operating area. Since ARIES waypoint lists do not provide for station-keeping, translation of the Loiter behavior utilizes multiple waypoints that maintain the assigned position. Additionally, waypoints must be recalculated and reissued for the duration of the monitoring period and the pattern must be interrupted when the period ends regardless of the status of the currently ordered waypoints.

Summarized results for virtual environment ERBM-controlled ARIES missions with a single MonitorTransmissions goal are provided in Figures 8.10. In the depicted mission, the ERBM controller directed the vehicle around a circular avoid area to the center of the rectangular operating area. Upon reaching the operating area, the vehicle proceeded to a depth of one meter (ARIES does not possess a bottom-mounted rudder and cannot effectively maneuver on the surface), slowed to conserve power, and

commenced a pattern of four waypoints arranged in a 30 meter square. Since the pattern was completed prior to the end of the monitor period, the controller reissued the loiter pattern waypoints multiple times. However, the final pattern was not completed prior to the end of the monitor period (the vehicle was transiting from the first to the second waypoint in the pattern), so the final sequence was interrupted and the vehicle was directed to the agenda-defined recovery position. As of the time of this writing, no corresponding in-water experiment has been conducted.

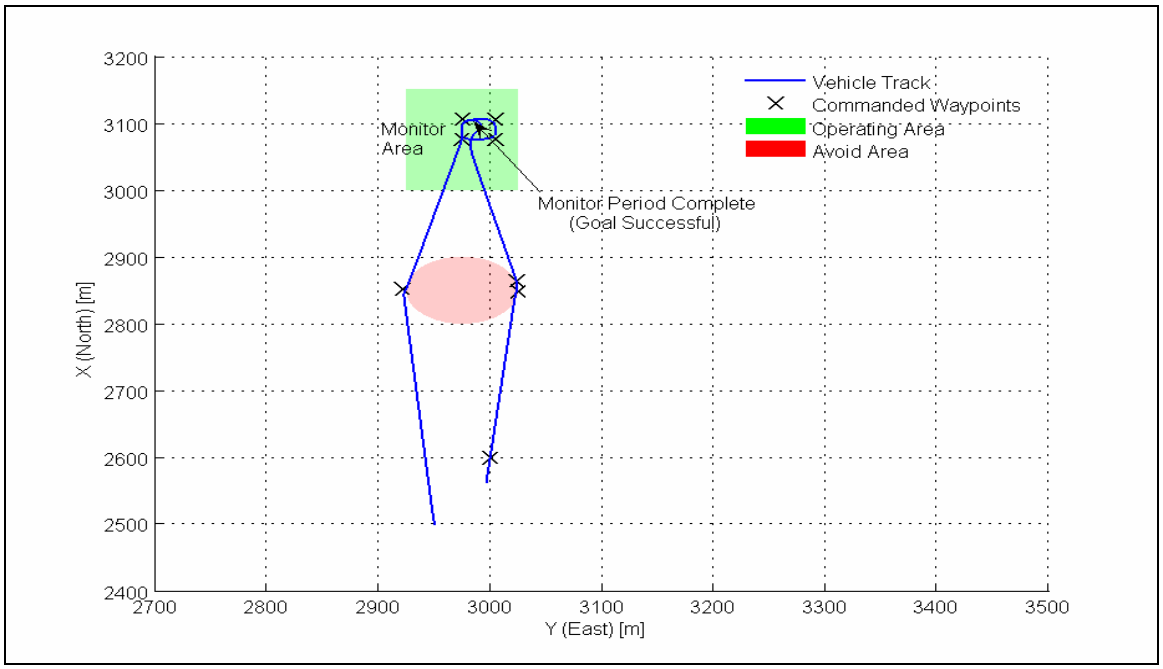


Figure 8.10. ARIES UUV Simulation Results for an ERBM-Controlled Mission with a Single MonitorTransmissions Goal and a Single Avoid Area

Remaining goal types for which in-water or virtual environment experiments were conducted—Search, Patrol, and SampleEnvironment—rely on search patterns generated according to the decision tree of Figure 6.7 to provide for uniform coverage of the operating area. Figures 8.11 and 8.12 provide summaries for virtual environment and in-water runs of an area-search mission with the potential for multiple targets. In both cases, the controller directed the vehicle around the circular avoid area and commenced a parallel-track search of the rectangular operating area. Location of the search target on the second leg of the pattern met part of the criteria for goal success. However since the goal specification indicates the potential for multiple targets, the

pattern was allowed to complete without interruption. Following completion, the controller directed the vehicle to the recovery point, again bypassing the depicted avoid area. Differences between virtual environment and in-water results are due to slightly different launch and recovery positions, navigation adjustments following the into-area and out-of-area GPS fixes of the in-water run (depicted in Figure 8.12), and noticeably less responsive left-turn performance at the end of the second search-pattern leg. Anomalies notwithstanding, the ERBM controller performed as advertised and directed the vehicle into and out of the operating area while bypassing the depicted avoid area and commanded a predictable and effective search pattern.

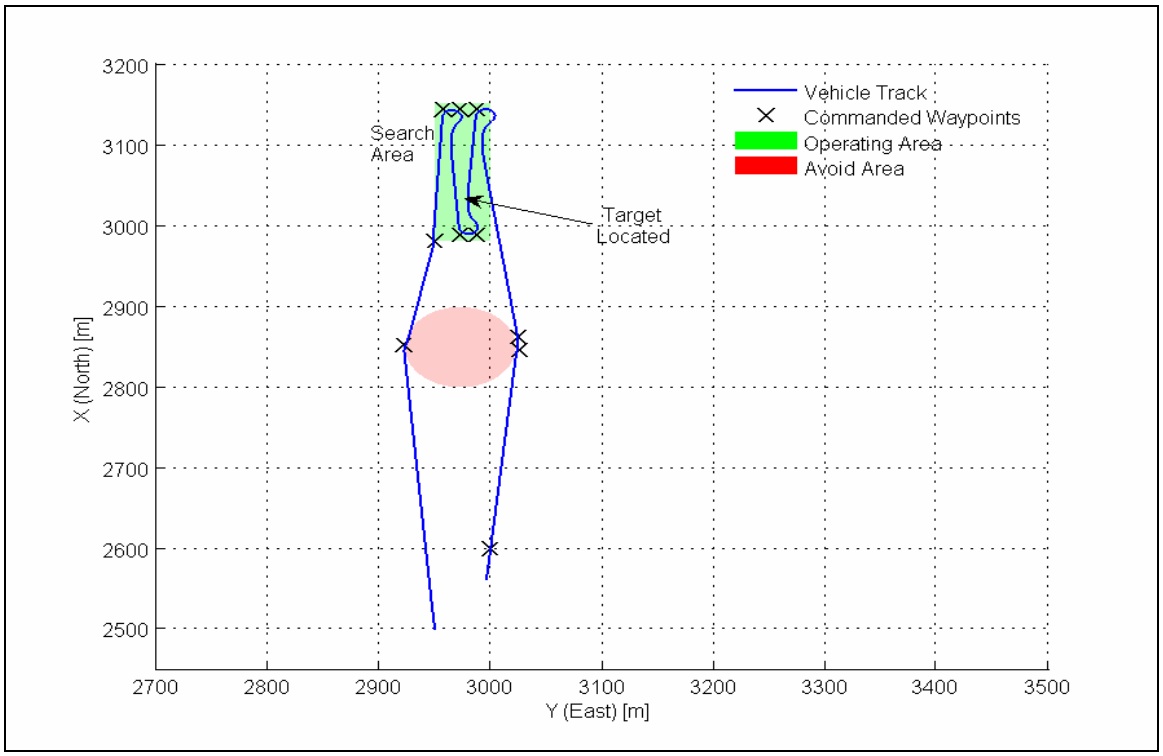


Figure 8.11. ARIES UUV Simulated Results for an ERBM-Controlled Mission with a Single Area-Search Goal with Potentially Multiple Targets

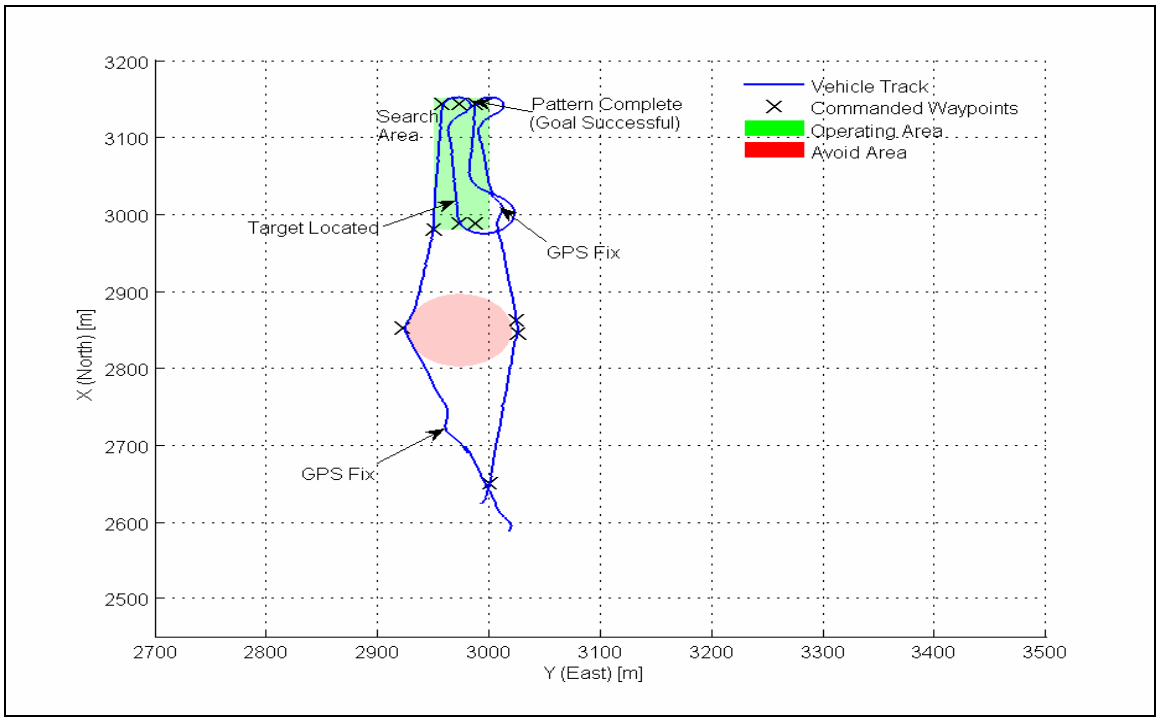


Figure 8.12. ARIES UUV In-Water Results from Monterey Bay (25 July 2006) of the Multi-Target Area-Search Mission of Figure 8.12

Figures 8.13 and 8.14 depict virtual environment and in-water mission summaries for a single-target search of the same operating area as the multi-target search depicted in Figures 8.11 and 8.12. As with the multi-target search, the ERBM controller directed the vehicle into and out of the operating area while bypassing the depicted avoid area. Once in the area, the decision tree called for the same parallel track pattern used in the multi-target search. However, in this case, the goal was immediately considered successful upon location of the target, so the pattern was interrupted and the vehicle proceeded to the recovery point. As in the multi-target search example, differences between the virtual environment and in-water can be attributed to slightly different launch positions and navigation adjustment following GPS fixes during the in-water run depicted in Figure 8.14. These differences, however, do not relate to the ERBM controller and do not effect the assessment of its performance. Thus, these examples provide a suitable demonstration of the ability of the ERBM finite-state-machine-based controller to react to changes in current goal status while directing overall mission flow. As in the multi-target example, the ERBM controller performed effectively both in simulation and in the real-world experiment.

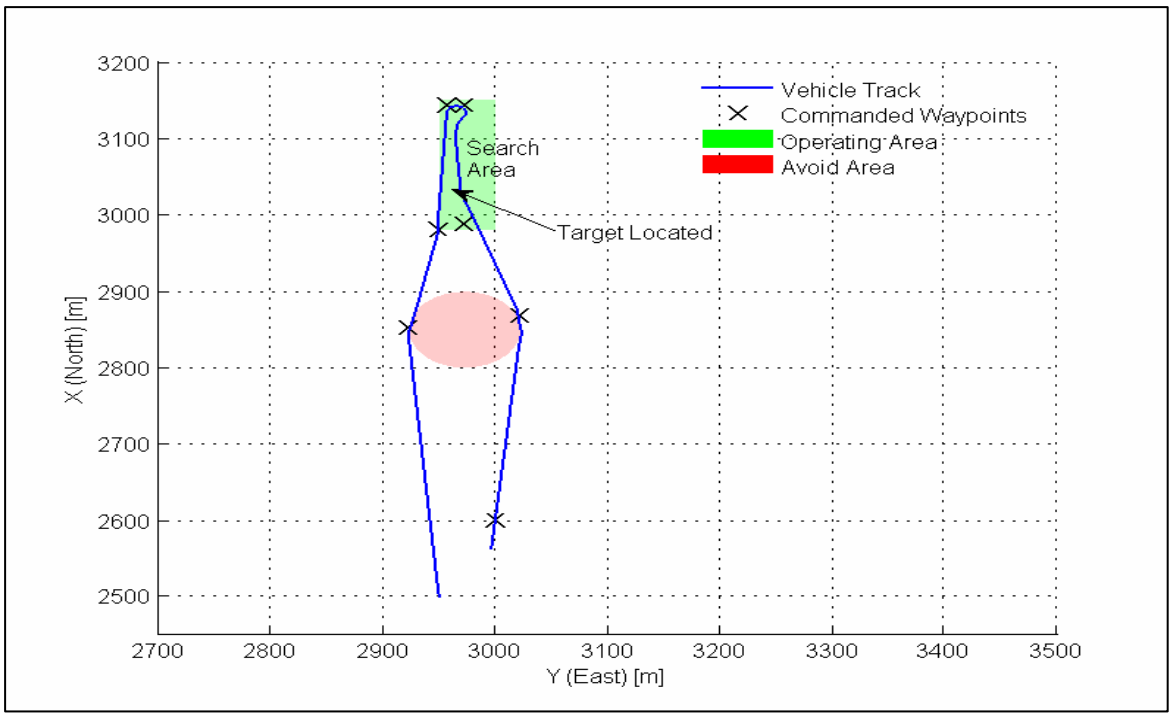


Figure 8.13. ARIES UUV Simulated Results for an ERBM-Controlled Mission with a Single Area-Search Goal for a Single Target

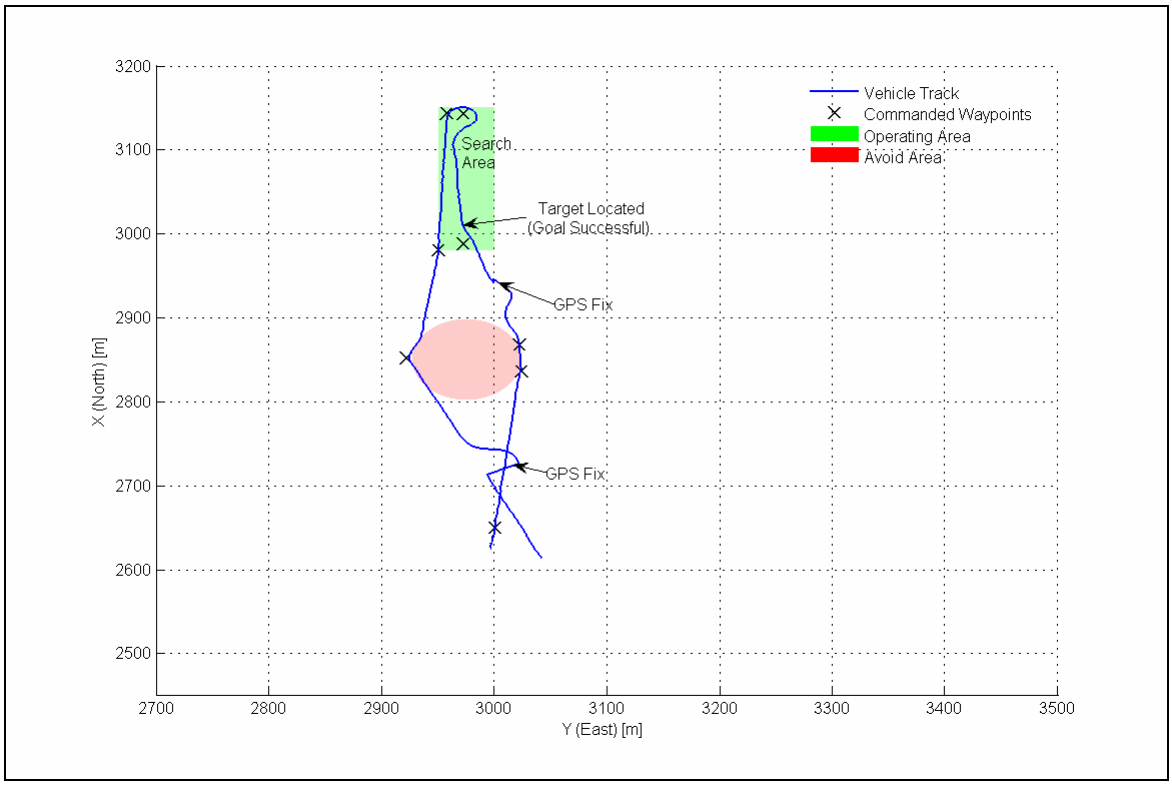


Figure 8.14. ARIES UUV In-Water Results from Monterey Bay (25 July 2006) of the Single-Target Area-Search Mission of Figure 8.14

Experiments with multiple-goal agendas were conducted only in simulation because of the extended mission run-times. All documented examples utilized the same mission which consisted of three goals and a single avoid area. The mission first called for a single-target area-search. If the target was located, the vehicle was to patrol a polygonal operating area. If the search failed to locate the target, the vehicle was to proceed to a circular operating area for a MonitorTransmissions goal. Success or failure of the MonitorTransmissions or Patrol goal indicated mission completion.

In the experiment depicted in Figure 8.15, the vehicle was directed to the search area and commenced a parallel-track pattern. Following the location of the search target on the second leg of the search (goal successful), the vehicle proceeded to the patrol area. The irregular shape of the area dictated a traveling-salesman-problem-based search pattern generated using the simulated annealing algorithm described in Chapter VI. The pattern was completed before the patrol period ended, so a second pattern was generated and commenced. However, the patrol period ended (meeting the criteria for goal success) shortly after the second pattern was begun. Thus, the pattern was interrupted and the vehicle transitioned to the designated recovery point.

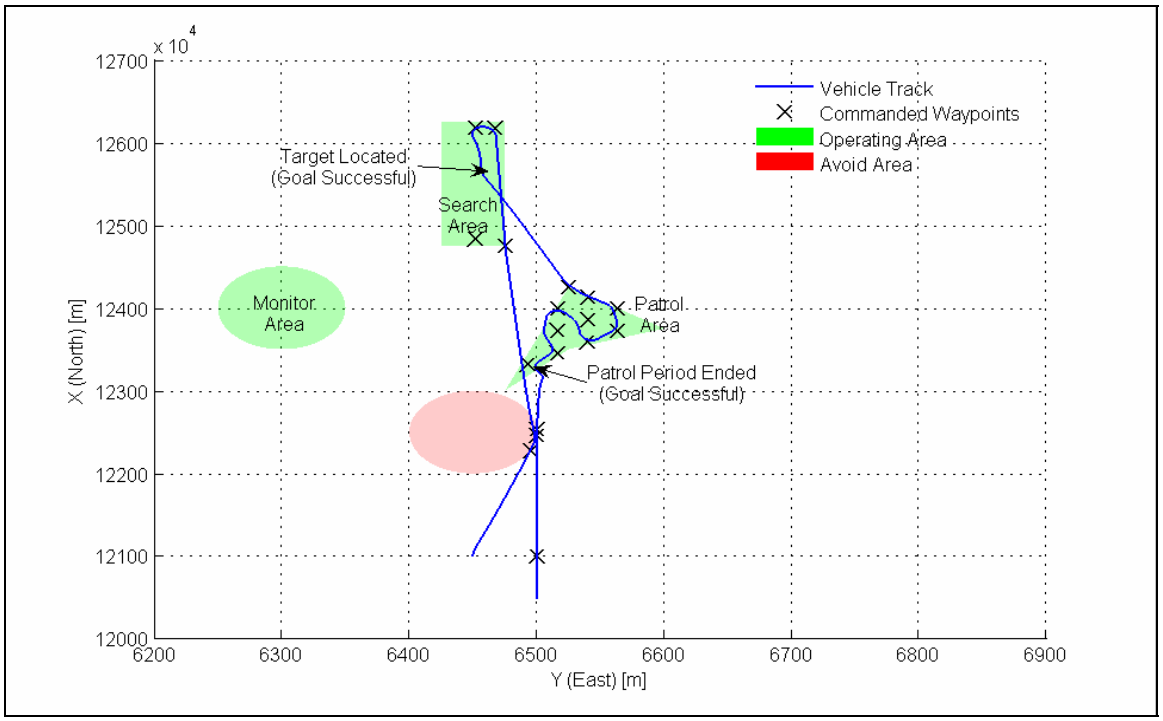


Figure 8.15. ARIES UUV Simulated Results for an ERBM-Controlled Mission with a Successfully Executed Area-Search and Patrol Goals

As with the preceding example, the mission summarized in Figure 8.16 began with a parallel-track search of the rectangular search area. However, in this instance the search target was not located, so the area-search goal was unsuccessful. The vehicle then transitioned to the operating area of the MonitorTransmissions goal. Upon arrival in the area, the ERBM controller directed the vehicle to slow, change depth to one meter, and loiter near the center of the operating area for the duration of the monitoring period (again, translated for ARIES as a 30-meter square waypoint pattern. Upon goal success (i.e., the end of the monitoring period), the vehicle was directed to the designated recovery position.

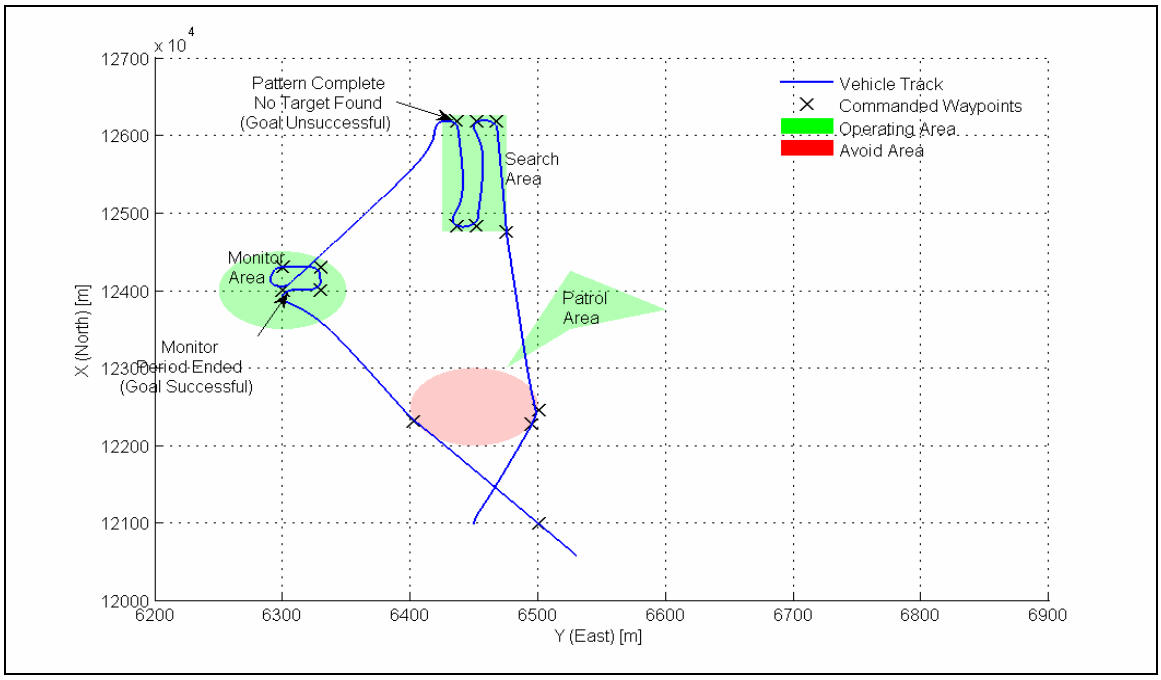


Figure 8.16. ARIES UUV Simulated Results for an ERBM-Controlled Mission with an Unsuccessful Area-Search Goal and a Successful MonitorTransmissions Goal

The successful execution of these multi-goal agendas provides a demonstration of ERBM control as the vehicle progresses through the mission-level state machine. Both goal-success and goal-failure transitions were executed and the vehicle was controlled in accordance with the mission definition in both cases.

D. SUMMARY

Only limited experimentation specifically focused on translations between AVCL and vehicle-specific data formats is documented in this chapter. However, the techniques

and data mappings described in Chapter V did prove sufficient for all tests, particularly if the use of potentially ambiguous open-loop behaviors was avoided and care was taken to ensure geographic origin consistency over the course of multiple translations. Additionally, translation from AVCL to a vehicle-specific format is an inherent aspect of on-vehicle ERBM implementation, so successful ERBM experimentation with actual vehicles implicitly demonstrates this capability.

The suitability of the ERBM control architecture for UAV and USV control was demonstrated in simulation using AUVW physically-based models and vehicle-control software. Experiments with the ARIES UUV, on the other hand, provide an on-vehicle exemplar indicative of the broader applicability of the ERBM architecture and its underpinning common autonomous vehicle data model.

ERBM experiments included missions containing AVCL Reposition, Search, Patrol, MonitorTransmissions, and SampleEnvironment goals. Of the remaining goal types, the vehicle-rendezvous implementation of (Nicholson, 04) demonstrates the viability of high-level Rendezvous goal control using an architecture along the lines of the ERBM. In fact, the (Nicholson, 04) control architecture characteristics are heavily leveraged in the overall ERBM design and provided a conceptual starting point for much of the work documented in Chapter VII. Jam and IlluminateArea goals are not typical of the types of goals likely to be required of a vehicle along the lines of the ARIES UUV, so on-vehicle experiments were not conducted for these goal types. They are potentially applicable to UAVs and USVs and tests have been run for these goal types using the AUVW. In the conduct of these missions, appropriate mission-system operation (i.e., jammer or illuminator) was assumed since these systems cannot be explicitly controlled using the existing AVCL task-level behavior set. Finally, the Attack, Decontaminate, Demolish, and MarkTarget goal types require a level of mission-system control not currently available in AVCL's task-level behavior set. Verification of the finite-state-machine-based control of these goal-types, therefore, remains an area for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

IX. CONCLUSIONS AND RECOMMENDATIONS

A. RESEARCH CONCLUSIONS

The overarching hypothesis explored in this work is that despite their apparent differences, autonomous vehicles of a particular type have enough similarities to facilitate the development of a common data model capable of expressing tasking, messaging, and mission results for arbitrary vehicles. Further it is surmised that this data model can be directly applied to actual vehicles in a variety of ways that potentially improve interoperability and foster the development of vehicle-independent support systems.

Perhaps the most straightforward conclusion to be drawn is that there is enough commonality between various vehicles to enable the implementation of a single data model suitable for the representation of arbitrary vehicle tasking, messaging, and mission results and that XML Schema provides a suitable mechanism for formal definition of this data model. XML has a number of advantages over bit-mapped binary or non-XML text-based formats that aide in the definition and use of this common data model including content governance, content verifiability, readability and platform independence. Further, XML documents complying with a well-designed XML schema are largely self documenting making them easier to work with when archiving or analyzing data. Finally, the ease with which XML can be incorporated into applications facilitates the development of applications that enforce correctness by abstracting the end user from the data model's syntactic and structural requirements.

Of primary importance in the definition of a common autonomous vehicle data model is the design of an appropriate vehicle tasking mechanism. This work defines a set of task-level behaviors that prove effective for this purpose. Additionally, the development of rigorously defined behavior activation and termination criteria proved important aspects of the task-level behavior definition process. Given an appropriate set of task-level behaviors and deterministic activation and termination criteria, any vehicle activity can be unambiguously represented within the constraints of the common data model. This work makes significant strides towards this end, particularly where vehicle

motion is concerned. Not surprisingly, virtually every subsequent aspect of this research relies upon a well-defined task-level behavior set.

A corollary hypothesis of this work was that it mechanisms can be implemented to support automated translations between the data model and vehicle-specific data formats. Thus, a second important result of this work is the development of techniques for translating between an XML-based data model and vehicle-specific tasking, messages, and results data. A preliminary requirement to actual translation is the development of mappings between the common data model and vehicle-specific formats. By the use of exemplars, this work demonstrates the viability of mapping between various vehicle-specific data formats and a common autonomous vehicle data model and identifies a number of issues associated with these mappings and the translations that they support.

XSLT is the obvious choice for converting data-model compliant data to text-based formats as indicated by its routine use in this role in a numerous of domains. However the use of XSLT extensions, development of a simulated XSLT mutable variable pattern, and the addition of a MetaCommand behavior to the task-level behavior set was required to support these translations. Although XSLT was quickly ruled out as a potential mechanism for the translation of vehicle-specific text into model-compliant XML the use of context-free grammars as translation mechanism has been explored in other contexts (e.g., natural language parsing). Their use in translating vehicle-specific data to model-compliant XML format proved a natural extension of this application. Similarly, the usefulness of XML encodings of a binary formats is becoming increasingly common. Two examples are provided by the development of XML encodings for the Distributed Interactive Simulation (DIS) protocol (McGregor, et al., 06) and JAUS messages (JAUS, 06). However, these other efforts utilize the XML encodings as a convenient form for working with the binary data rather than as an intermediate form supporting ultimate translation using XSLT making the application explored in this work noteworthy.

The development of a declarative means of task-specification within the data model results from a desire to provide a more abstract and intuitive method of mission

definition than is provided by scripting. The success of a task-specification mechanism of this sort is implicitly based on the hypothesis that these declarative missions can be effectively converted into task-level behavior scripts. Previous work with layered control architectures suggests methods similar to those explored in this work for generating increasingly detailed command sequences at lower layers of the architecture. However, the goals of an AVCL declarative mission are more abstract in nature than those of typical layered architectures, making the task of generating task-level behavior sequences from AVCL goals more difficult. In particular, the use of search and planning algorithms, most notably as the simulated-annealed traveling salesman problem algorithm, to generate area-coverage patterns (i.e., search patterns) is significant.

On its face, the inference of declarative goals from task-level behavior sequences might not seem a requirement for successful application of a common autonomous vehicle data model. This form of translation is, however, required in order to meet the goal of interchangeability of any tasking form and facilitates integration of the common data model into broader command and control systems for which typical autonomous vehicle script-level command is not always meaningful. Based upon the premise that scripts that are intended to accomplish certain types of goals are potentially identifiable by identifiable characteristics, the work here concerning the inference of intent from task-level behavior scripts is fairly unique. Nevertheless, it is ultimately a requirement for the effective use of the common data model beyond the domain of autonomous vehicle operations. Relying on the identification of suitable characteristics as well as the implementation of an actual script-classification mechanism, the case-based reasoning and naïve Bayes classification systems developed in the course of this work provide an initial capability. Although surprisingly successful in their current form, they are considered a starting point for future work in this area.

When combined, the translation mechanisms discussed in the preceding paragraphs demonstrate the general interchangeability of vehicle tasking types. As indicated in Figure 9.1, vehicle tasking in any form, whether vehicle-specific or constrained by the data model, is potentially convertible to any other form.

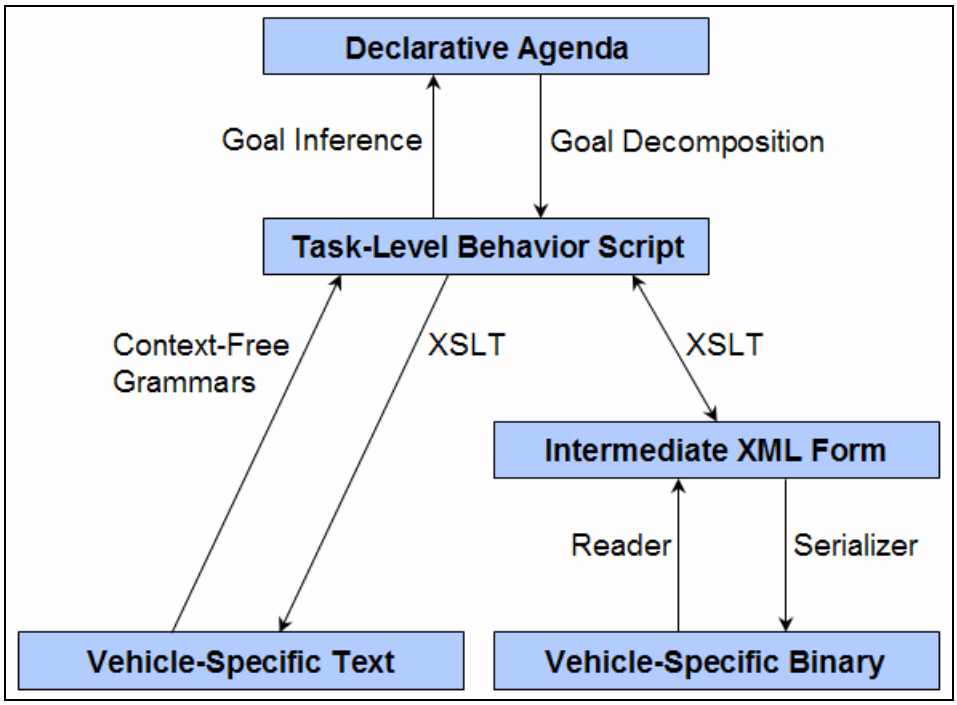


Figure 9.1. Mechanisms Supporting Autonomous Vehicle Tasking Form Interchangeability and Automated Translation between Forms

A final hypothesis explored in this research is based on the observation that generation of task-level behavior scripts for accomplishing declarative goals is conceptually similar to the goal decomposition occurring at the upper levels of a multi-layer vehicle-control architecture. It might be possible, therefore, to implement a multi-layer control architecture around this functionality. Further, the ability to generate vehicle-specific commands from task-level behaviors implies that this architecture can be designed to augment a vehicle's existing controller. The result of exploration and experimentation with this hypothesis is the ERBM control architecture.

Although similar in purpose, the common-data-model-based ERBM differs from other vehicle-independent command efforts in implementation. Developmental vehicle-independent command languages typically rely on target vehicle implementation of the common format (e.g., C2L) or programmatic generation of vehicle-specific commands using undisclosed mechanisms (e.g., CCL). The ERBM avoids these potentially cumbersome requirements by making vehicle-specific command generation a function of the controller itself and leveraging the translation mechanisms developed in the course of

this work to generate suitable command sequences that accomplish high-level goals. This facilitates on-vehicle implementation by minimizing required modification of existing vehicle control software. Thus, the ERBM implementation explored here is noteworthy in its method of realizing vehicle-independence.

B. RECOMMENDATIONS FOR FUTURE WORK

Overall, this research successfully addresses the objectives defined in Chapter I. However, as is typically the case with research of this sort, much remains to be accomplished before this work can be effectively applied to operational systems. Additionally, the concepts explored here might be improved, augmented, or extended in a number of ways to broaden their applicability.

An obvious potential addition to this work involves the application of these concepts to more actual vehicles. The mappings and translations between AVCL and JAUS messages, REMUS objectives, Seahorse commands and Phoenix behaviors are correct according to the available references. They have not, however, been tested in an operational environment with actual vehicle software. Experiments of this sort will further verify the assertions of this work across a broader spectrum.

In a similar vein, a potential extension of the task-level behavior set to more fully meet the requirements of UGVs, USVs, and UAVs merits attention. With the exception of the JAUS message set, all of the data formats experimented with in this research apply only to UUVs. A rigorous analysis of existing command and message formats for other vehicle types will facilitate the development and implementation of behaviors appropriate for these types of vehicle and enable the use of a common autonomous vehicle data model with a broader array of vehicle types. Ultimately, a task-level behavior extension of this sort will be required before the data model will provide for true compatibility between both dissimilar vehicles of the same type and vehicles of different types.

As discussed briefly in Chapter VI, the task-level behavior set developed in this work does not provide for control of mission systems, payloads or manipulators. Clearly, this limits the types of vehicle activities that can be controlled. From the standpoint of the common data model, five types of declarative goals cannot be explicitly converted to task-level behaviors at all and the specific objectives of the remaining types (e.g.,

frequencies of interest for a MonitorTransmissions goal) require the use of MetaCommand behaviors that may or may not be translatable to vehicle-specific commands. Thus, the development of a broadly applicable mission-system interface might be the most important requirement for full realization of the potential of a common autonomous vehicle data model. This is especially true of the high-level planning and control applications of Chapters VI and VII. Fortunately, an interface of this sort is also a priority of a number of other robotics development efforts and early results are already becoming available. The JAUS message set, for instance, includes commands for manipulating jointed manipulators and video devices (JAUS, 04-3) and a U.S. Navy-sponsored effort is expected to release a UUV-payload interface standard later this year (ASTM, 06). Regardless of which standards are ultimately accepted, incorporation of their functionality into the data model's task-level behavior set will allow for more explicit vehicle control. Further, this increased level of control will be available whether vehicle activity is directed by a predefined task-level behavior script or those scripts are generated by the upper layers of a hybrid control architecture (i.e., ERBM).

The ERBM controller and the planning algorithms upon which it relies also provide a number of potential areas for future work. The search planning algorithms described in Chapter VI, for instance, are implemented to develop a single-vehicle, single-pass pattern. That is, the waypoints making up the pattern direct a single vehicle to cover the objective area exactly once. Since support of multi-vehicle operations is a design objective of the common autonomous vehicle data model, it stands to reason that multi-vehicle planning algorithms should be an inherent part of a high-level controller based on the data model. A planner of this sort might divide the area into subsections with a different vehicle assigned to each, or it might plan whole-area search patterns for all participating vehicles.

Similarly, even single-vehicle searches might benefit from the development of plans that prescribe multiple passes over the search area (possibly at different depths or altitudes for UUV or UAV searches). Using Equation 6.1, a search track-spacing of 1.333 times the sensor-sweep width is required to obtain a probability of detection of 0.75 for a single pattern search while a track-spacing of 4.0 times the sensor-sweep width will only provide a probability of detection of 0.5. However, based on the combinatorial

mathematics inclusion-exclusion principle (Mendenhall, et al., 01), two searches of an area, each with a probability of detection of 0.5 will also provide an overall probability of detection of 0.75. This raises the possibility that a widely spaced multi-pass search might provide for the same probability of detection as narrowly spaced single-pass search with a shorter overall travel-distance.

Among the richest potential areas for future work relating to this research is the ERBM Tactical level. Since implementation efforts documented in this dissertation focused on Strategic-level planning and task-level behavior translation for issue to the Execution level (i.e., the existing vehicle control system), there is ample opportunity for improvement at the ERBM Tactical level. Many more general autonomous vehicle research efforts that seemingly fall outside the scope of the common data model are potentially applicable at the ERBM Tactical level. For instance, object detection and classification, localized path planning and obstacle avoidance, feature-based navigation, simultaneous localization and mapping, and onboard systems monitoring and associated fault detection and response are all appropriately implemented at the Tactical level as described in Chapter VII. A corollary requirement brought about by the implementation of these and other Tactical-level capabilities will be the development of supporting inter-level messages (in addition to those of listed in Table 7.3) to take advantage of new functionality. Ultimately, adding ERBM Tactical-level capabilities will facilitate the evolution of the Strategic level goal-specific finite state machines and replanning capabilities.

At least two potential areas for future work relating to script-intent inference (i.e., assigning an appropriate declarative goal to a task-level behavior sequence) are easily identified. The results documented in Chapter VI for both the case-based reasoning and naïve Bayes classifiers are encouraging. However, a more in-depth study of the relationship of various script characteristics to the declarative goal types, utilization of a larger recall set, and exploration of other machine learning techniques might provide still better results. In particular, the relatively low PointSearch recall (i.e., the proportion of PointSearch scripts that were identified as such) for both planners provides room for improvement.

Additionally, the performance comparison between the case-based reasoning and naïve Bayes systems clearly indicates that the performance of both systems are more accurate in identifying certain types of missions than others. The performance of both systems might benefit from analysis of receiver operating characteristics curves (Montgomery and Runger, 03) correlating to a Boolean identification for each goal type. Using the case-based reasoning distance metric or computed naïve Bayes *a posteriori* probabilities, the rates of true and false positives and negatives can be determined as a function of a minimum classification threshold for each goal type. The ultimate classification, then, can be based on the curve for which a false positive is least likely given the actual distance or computed probability. This classification may or may not be the same as would be made based solely on the minimum distance or maximum *a posteriori* probability. Receiver operating characteristic curves might also be used to determining a maximum acceptable distance or minimum probability for each goal type (based on the resultant false positive rate) if it is determined that it is more desirable to fail to classify a mission than to assign it an incorrect classification.

Although all specific objectives of this research have been addressed, the use of the common data model to directly support coordinated operations is not directly demonstrated. Rather, the assumption is made that if message data can be translated between vehicle-specific formats using the common data model, vehicles will be able to interpret and respond to messages from other vehicles. Demonstration of this capability using actual or simulated vehicles is the obvious next step. If the ERBM is used to provide high-level control for all participating vehicles, data model facilitation of coordinated operations is obvious since inter-vehicle communications is implemented at data-model-dependent Tactical and Strategic levels. In this case, translation of AVCL messages to vehicle-specific formats is not required and the ERBM controllers can potentially provide for coordinated operations even among vehicles that are not designed to operate as part of a multi-vehicle system. In cases where the ERBM controller is not used, but the participating vehicles possess inter-vehicle communications capability, translations between the native formats will be required to achieve any level of autonomous coordination.

A further question remains to be addressed for any multi-vehicle system requiring translation between vehicle-specific message formats (i.e., a system in which one or more participating vehicles does not implement the data model directly). That is, where are the data format translations most appropriately conducted? A first possibility is to require all transmitting vehicles to convert messages to the destination vehicle's native message format prior to transmission (i.e., translation occurs on the transmitting vehicle). A second possibility is to allow transmitting vehicles to send messages using their own native format. Receiving vehicles might then be required to translate the message to their own native format upon receipt (i.e., translation occurs on the receiving vehicle). A final possibility is to allow transmitting vehicles to send messages using their own native format and using an intermediate server to translate and retransmit the message in the destination vehicle's message format (i.e., translation and retransmission occurs on an intermediate server). Additionally, the use of these methods might be combined and tailored to the capabilities of the individual vehicles. Each of these potential translation paradigms has potential advantages and disadvantages, so study and experimentation with various configurations and operating environments is warranted.

A peripheral follow-on to this research might involve the actual incorporation of the common data model into a larger command and control system. Alluded to in a number of contexts thus far, this capability is a requirement for the effective integration of autonomous vehicles into larger scale operations. This task is potentially simplified by the fact that the declarative goal-types available in AVCL are intentionally aligned with the JC3IEDM Action-Tasks considered most appropriate for autonomous vehicle execution. Incorporation of AVCL data into a JC3IEDM system will require the translation of data-model documents into business objects implemented by the JC3IEDM system and vice versa. Since business objects typically take the form of XML documents, XSLT stylesheets based on data mappings similar to those described in Chapter V are the most appropriate mechanisms for conducting both of the required translation.

Another area of potential related research involves the investigation of data model use in the design and implementation of autonomous vehicle support system interfaces. As discussed briefly in Chapter III, the ease with which XML can be processed facilitates

the development of applications that effectively distance the end user from the intricacies of the data model. Given the increasing emphasis on human-computer interaction aspects of unmanned vehicle systems, this is a potentially important data model application. In fact, the straightforward realization that end users are likely to possess more mission area expertise than autonomous vehicle expertise, makes the fielding of intuitive systems for vehicle programming, monitoring and analysis crucial to the overall success of vehicle systems in operational environments. The AUVW provides evidence that a vehicle-independent data model defined as an XML vocabulary can facilitate the development of support systems that are both operationally robust enough to support mission requirements and user friendly enough to be utilized by lay operators.

The potential influence of a common data model on the design of user interfaces for autonomous vehicle programming highlights a more subtle question that bears exploration—what is the appropriate level of mission-programming capability of an operational autonomous vehicle support system? Interface simplicity and intuitiveness notwithstanding, it is probably undesirable to provide a lay user with the completely unrestricted mission definition capability. In many instances, it might be sufficient to encode rules within the application to preclude mission-definition errors (e.g., a loop in the mission-level state machine of a declarative agenda). In other cases, the potential risks posed by a mission programming error (e.g., vehicle loss or attacking the wrong target) might justify significant restrictions to the mission-definition process. These might include the required use of built-in preplanned missions or required verification in simulation prior to execution in an actual vehicle. Ultimately, a rigorous analysis of the potential risks associated with expected operations (i.e., operating environment, tasking, operator expertise, etc.) and available safeguards is required to address this issue.

Finally, as discussed in Chapter IV, even though the data model developed in the course of this work provides all of the functionality required to meet the objectives of this research, it is not semantically rich enough to be accurately classified as an ontology. Given the increasing capabilities demonstrated by ontologies and Semantic Web applications in other domains, it is possible that a more ontological data model might provide advantages in this problem area as well. Further exploration of the capabilities of

ontologies, Semantic Web applications, and how they might be applied to the autonomous vehicle domain might prove a fitting extension to this work.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: THE AUTONOMOUS VEHICLE COMMAND LANGUAGE (AVCL)

A. INTRODUCTION

AVCL is a schema-governed XML vocabulary intended for use in defining autonomous vehicle tasking, exchanging messages between vehicles, and encoding vehicle mission results in a meaningful way. This appendix provides a detailed summary of the AVCL model's content, structure and semantics. The data model defined by the AVCL schema has evolved to support the work described here. It is entirely likely that this evolution will continue as this work proceeds. Thus, the data-model description provided here applies to AVCL in its current form and may not be equally applicable to future implementations.

The description begins with a summary of schema-defined simple types (i.e., types that are defined as restrictions of existing XML primitive or derived types). This is immediately followed by a description of schema-defined complex types that are reused in various places in the schema. After the data-type discussion, the structure and content of AVCL contents is described beginning with a description of each available document type (i.e., valid root-level tag) and high-level document structure. Finally, the structure, content and implied semantics of documents used for mission definition, mission results encoding, and inter-vehicle messaging is discussed.

Finally, it must be noted that AVCL is a developmental vocabulary. As such a number of components have not been fully implemented. Additionally, occasional inconsistencies or undefined elements may be encountered. In general, these pertain to peripheral or infrequently required elements or portions of the schema associated with vehicles or operations that were not exhaustively explored in the conduct of this research.

B. SIMPLE DATA TYPES

1. Numerical Data Types

Table A.1 lists and describes numerical types defined in the AVCL. Numerical data that requires range constraint All numerical data in an AVCL document requiring constraint beyond that provided by the XML numerical types defined in (W3C, 04-3)

uses one of these types. The int, double, and unsignedShort predefined types are also used extensively within the AVCL schema.

AVCL Type	XML Base Type	Range	Description
positiveScalarType	double	>0	Any positive double-precision floating point value.
nonNegativeScalarType	double	>= 0	Any non-negative double-precision floating point value.
positiveIntType	int	>0	Any positive 32-bit integer.
nonNegativeIntType	int	>= 0	Any non-negative 32-bit integer.
percentType	double	[0..100]	Indicates a percentage of an unspecified maximum.
signedPercentType	double	[-100..100]	Indicates a signed percentage of an unspecified maximum.
calendarDaysType	unsignedByte	[1..31]	Enumeration for the day-of-the-month portion of a calendar date.
clockHoursType	unsignedByte	[0..23]	Enumeration for the hour portion of a wall-clock time.
clockMinutesOrSecondsType	unsignedByte	[0..59]	Enumeration for the minute or second portion of a wall-clock time.
timeZoneType	byte	[-12..12]	Enumeration for a time zone relative to Greenwich Mean Time .
latitudeType	double	[-90..90]	Any latitude value (positive indicates northern hemisphere).
longitudeType	double	[-180..180]	Any longitude value (positive indicates eastern hemisphere).
headingType	double	(0..360]	An orderable vehicle heading (degrees).
orientationType	double	(-360..360)	An angle (degrees) describing a vehicle Euler angle.
areaOrientationType	double	[-90..90]	Represents a rotation angle for a geographic area (positive indicates clockwise).
priorityType	int	[1..255]	Value defining a message priority (lower indicates higher priority).

Table A.1. AVCL Numerical Simple Types.

2. String Enumerations

The AVCL schema defines a number of string-based enumerations. As mentioned in Chapter IV, defining string types such as these is equivalent to using integer enumerations. The use of meaningful strings, however, instead of integers makes for more readable and intuitive documents. A description of available types, their valid values and their meanings follows.

- uuvCapabilityType: describes a capability that a UUV must possess in order to successfully complete a defined mission. Valid values:

  thrusterPowered: vehicle uses propellers to maintain forward speed.
  bodyThrustersInstalled: vehicle must possess cross body thrusters that enable vertical, lateral and rotational motion regardless of forward speed.
  hoverCapable: vehicle must be capable of stationary hovering.
  altitudeCapable: vehicle must be capable of maintaining a specified altitude above the bottom.
  gpsCapable: vehicle must be capable of GPS navigation.
  communicationsCapable: vehicle must be capable of run-time inter-vehicle or vehicle-control station communications.

- ugvCapabilityType: describes a capability that a UGV must possess in order to successfully complete a defined mission. Valid values:

  tracked: vehicle must be tracked vice wheeled.
  gpsCapable: vehicle must be capable of GPS navigation.
  communicationsCapable: vehicle must be capable of run-time inter-vehicle or vehicle-control station communications.

- usvCapabilityType: describes a capability that a USV must possess in order to successfully complete a defined mission. Valid values:

  gpsCapable: vehicle must be capable of GPS navigation.
  communicationsCapable: vehicle must be capable of run-time inter-vehicle or vehicle-control station communications.

- uavCapabilityType: describes a capability that a UGV must possess in order to successfully complete a defined mission. Valid values:

  fixedWing: vehicle must be fixed wing.
  rotaryWing: vehicle must be rotary wing (hover capable).
  multiEngine: vehicle must have more than one engine.
  gpsCapable: vehicle must be capable of GPS navigation.
  communicationsCapable: vehicle must be capable of run-time inter-vehicle or vehicle-control station communications.

- frequencyUnitType: specifies frequency units for an acoustic or electromagnetic transmission. Valid values:

  Hz: hertz.
  KHz: kilohertz.
  MHz: megahertz.
  GHz: gigahertz.

- turnDirectionType: specifies a turn direction. Valid values:

  left: turn is to be to the vehicle's left or port.
  port: same as left.
  right: turn is to be to the vehicle's right or starboard.
  starboard: same as right.

- trackModeType: specifies a waypoint homing mode. Valid values:

  directTo: always travel directly towards the goal location.
  trackTo: always correct to a specified track so as to approach the waypoint from a specific direction.

- monthsType: specifies a calendar month. Valid values:

  January, February, March, April, May, June, July, August, September, October, November, and December.

- datumType: specifies whether a search is to focus on a single point (operating area center) or provide for full area coverage. Valid values:

  point: the search is to focus on the centroid of the operating area.
  area: the search is to provide equal coverage of the entire operating area.

- illuminatorType: specifies a type of illuminator capable of providing area illumination. Valid values:

  pyrotechnic: flare or other pyrotechnic illumination source.
  spotlight: directable searchlight or spotlight illumination source.

- markerType: specifies a type of marker for location or object marking. Valid values:

  laser: laser designator marking.
  smoke: visible smoke marker.

- contaminantType: specifies a type of contaminant that is to be tested for or cleansed. Valid values:

  nuclear: radiological contamination or hazard.
  chemical: chemical weapon or agent contamination.
  biological: biological contamination or hazard.
  toxin: potentially poisonous or hazardous substance or chemical.
  explosive: explosive agent or component.

- weaponStatusType: specifies the conditions under which weapons can be employed. Valid values:

  safe: Weapons authorized in self defense or in response to a formal order.
  tight: Weapons authorized against targets positively identified as hostile.
  free: Weapons authorized against targets not positively identified as friendly or neutral.

- reportingCriteriaType: specifies when a vehicle is to make status reports while attempting to accomplish an agenda-mission goal. Valid values:

  never: do not make status reports.
  periodic: make status reports at specified intervals.
  statusChanged: make status reports when goal-execution status changes.
  onCommence: report when commencing execution of a goal.
  onComplete: report when completing execution of a goal.

- acknowledgeType: specifies the circumstances under which receipt of a message is to be acknowledged. Valid values:

  yes: always acknowledge receipt of this message.
  no: never acknowledge receipt of this message.
  optional: acknowledgement is permissible but not required.
  positiveOnly: only acknowledge this message if the requested information or action will be provided.
  negativeOnly: only acknowledge this message if the requested information or action will not be provided.

- informationRequestType: content for information request messages defining the type of information that is being requested. Valid values:

  ping: request for a simple presence message.
  sensorData: request for a sensor data report
  contactSummary: request for a summary of all current contacts.
  controlSettings: request for the receiving vehicle's currently ordered control settings.
  posture: request for the vehicle's current location and orientation information.
  velocity: request for the vehicle's current velocity information.
  waypoint: request for the vehicle's next destination location.

- vehicleGroupCompositionType: content for messages relating to the maintenance of cooperating groups of vehicles. Valid values:

  initiateGroupFormation: begin the group-formation process.
  finalizeGroupFormation: end group-formation and begin work.
  dissolveGroup: terminate the existence of a group and release all vehicles making up the group for other tasking.
  locateGroup: sending vehicle is attempting to make contact with an established group if one is present.
  joinGroup: sending vehicle is attempting to join an established group.
  leaveGroup: sending vehicle is leaving an established group.

C. REUSABLE COMPLEX DATA TYPES

AVCL defines a number of reusable complex types. These define element content models including child element and attribute names and types.

The most rudimentary AVCL complex type is the noValueElementType. Instantiated elements of this type have no child elements and are intended to convey information primarily through the assigned element name. These elements may include the optional attributes listed in Table A.2. These attributes, referred to as AVCL's "common attributes," are valid with any element in the AVCL tag set. Their use throughout the schema is assumed to comply with Table A.2 unless otherwise noted.

Attribute Name	Type	Use	Description
description	xsd:string	optional	Provides arbitrary amplifying information describing the element.
timeStamp	positiveScalarType	optional	Used to associate a time with the element.
id	xsd:ID	optional	Unique identifier that can be used elsewhere to reference this element.

Table A.2. Optional Attributes Available for use with all AVCL Elements

AVCL defines a number of complex types with a single data item. Described in Table A.3, these complex types use a "value" attribute to hold the data of interest. The attribute type is constrained using a built-in XML (indicated by the XML "xsd" namespace designator) or an AVCL simple type. A numericalBlockElementType is also available. This type uses required "minimum" and "maximum" attributes (nonNegativeScalarType and positiveScalarType) to specify a numerical range.

Complex Type	Value Attribute Type	Value Attribute Description
scalarElementType	xsd:double	A double precision numerical value.
positiveScalarElementType	positiveScalarType	A positive double-precision numerical value.
nonNegativeScalarElementType	nonNegativeScalarType	A non-negative double-precision numerical value
integerElementType	xsd:int	A 32-bit integer.
positiveIntegerElementType	positiveIntType	A positive 32-bit integer.
nonNegativeIntegerElementType	nonNegativeIntType	A non-negative 32-bit integer.
booleanElementType	xsd:boolean	A Boolean value.
stringElementType	xsd:string	An arbitrary string value.
tokenElementType	xsd:token	A white-space-free string value.
percentElementType	percentType	A percentage.
signedPercentElementType	signedPercentType	A signed percentage.
headingElementType	headingType	An orderable vehicle heading.
areaOrientationElementType	areaOrientationType	A rotation angle to be applied to a geographic area.
priorityElementType	priorityType	Specifies a message priority.
acknowledgeElementType	acknowledgeType	Specifies message acknowledgement requirements.
trackModeElementType	trackModeType	Specifies a waypoint homing mode.

Table A.3. AVCL Complex Types Containing a Single Data Item in the form of a "value" Attribute

Among the requirements of an autonomous vehicle data model is the ability to specify geographic positions. AVCL types and attributes available for this purpose are depicted in Figure A.1 and Table A.4. All AVCL positions derive from one of two types (xyElementType and latitudeLongitudeElementType) that encode a latitude and longitude or a Cartesian coordinate pair. AVCL extends these types to implement absolute and relative positions through the AbsoluteHorizontalPositionElements and HorizontalPositionElements groups of Figure A.1. These, in turn, form the basis of the horizontalPositionElementType and absoluteHorizontalPositionElementType complex types (not depicted) that consist of an element with a single absolute or relative position child element.

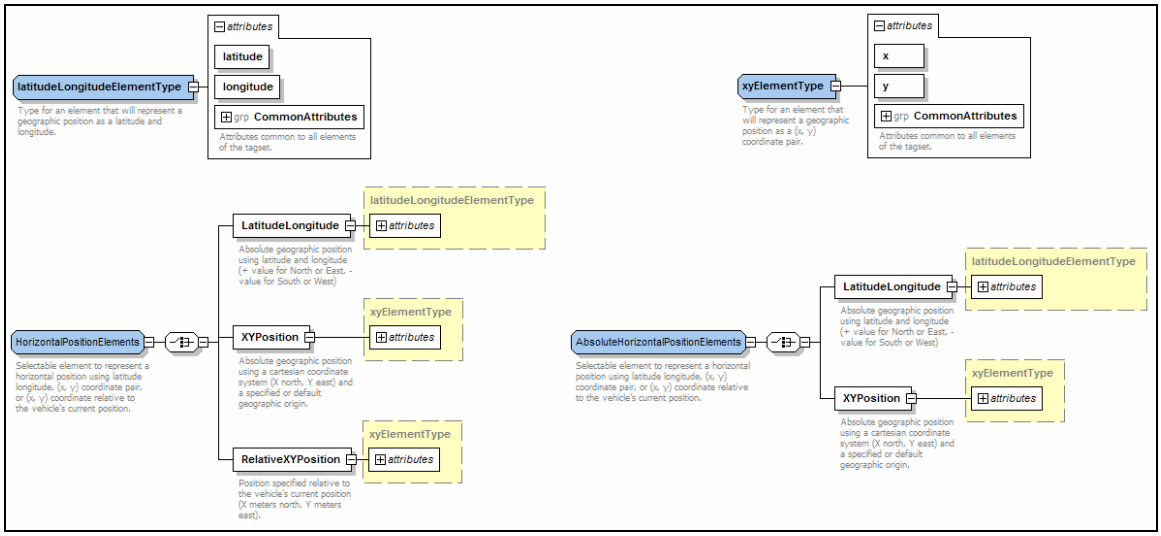


Figure A.1. AVCL Complex Types and Groups for Representing Geographic Position

Name	Type	Description
x	xsd:double	Specifies a Cartesian x coordinate in either the global coordinate frame or relative to the current vehicle position.
y	xsd:double	Specifies a Cartesian y coordinate in either the global coordinate frame or relative to the current vehicle position.
latitude	latitudeType	Specifies the latitude portion of a geographic position.
longitude	longitudeType	Specifies the longitude portion of a geographic position.

Table A.4. Attributes of AVCL Elements Available for Representing Absolute and Relative Positions

An obvious application of the AVCL types for specifying position is a set of types for specifying geographic areas. Areas in AVCL are specified as points, circles, rectangles, or polygons. A point consists of an element (with a "Point" name and the common AVCL attributes) with a single child (absoluteHorizontalPositionElementType). The structure and composition of the circleElementType, polygonElementType and rectangleElementType are depicted in Figures A.2, A.3, and A.4 respectively. The circleElementType is fairly self-explanatory and specifies the area with child elements containing the geographic center of the circle and a radius in meters. Polygons are specified simply with a sequence of either latitudeLongitudeElementType or xyElementType position elements containing the ordered vertices of the polygon. Rectangles are specified with child elements for the geographic position of the northwest corner, the horizontal width in meters, the height in meters, and an optional element for a clockwise angle of rotation in degrees. An AreaElements group is used to provide a container type for a single area element.

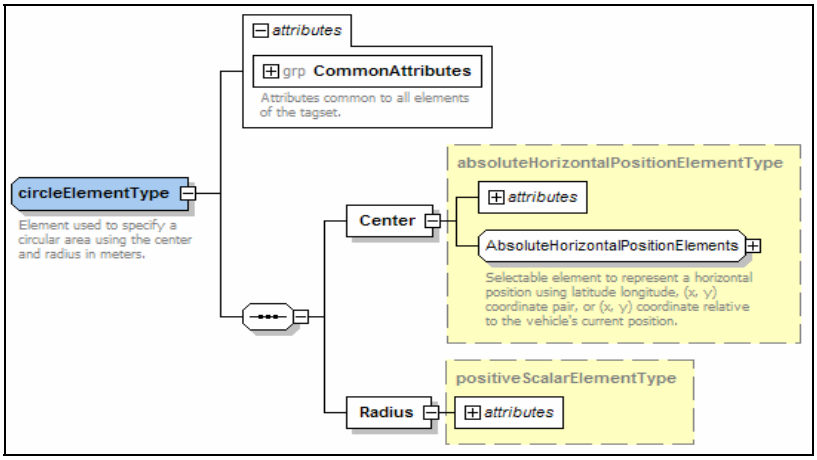


Figure A.2. The AVCL circleElementType for Specifying a Geographic Area as a Circle

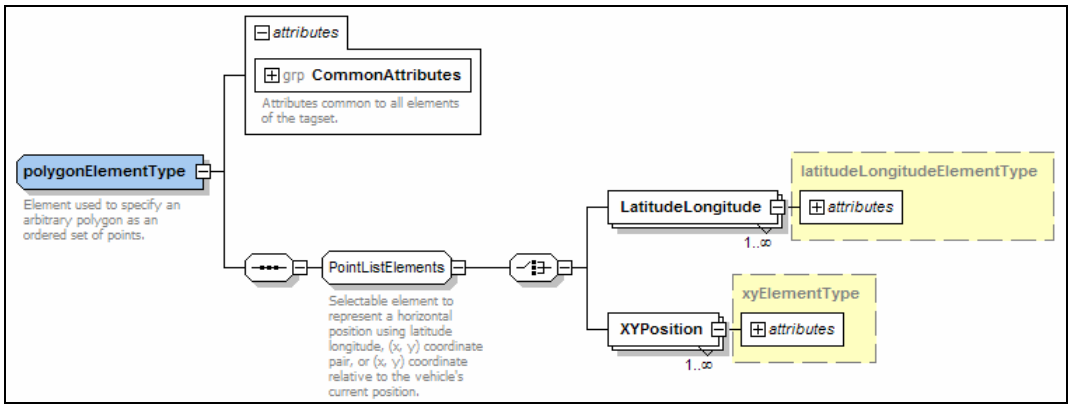


Figure A.3. The AVCL polygonElementType for Specifying a Geographic Area as an Arbitrary Polygon

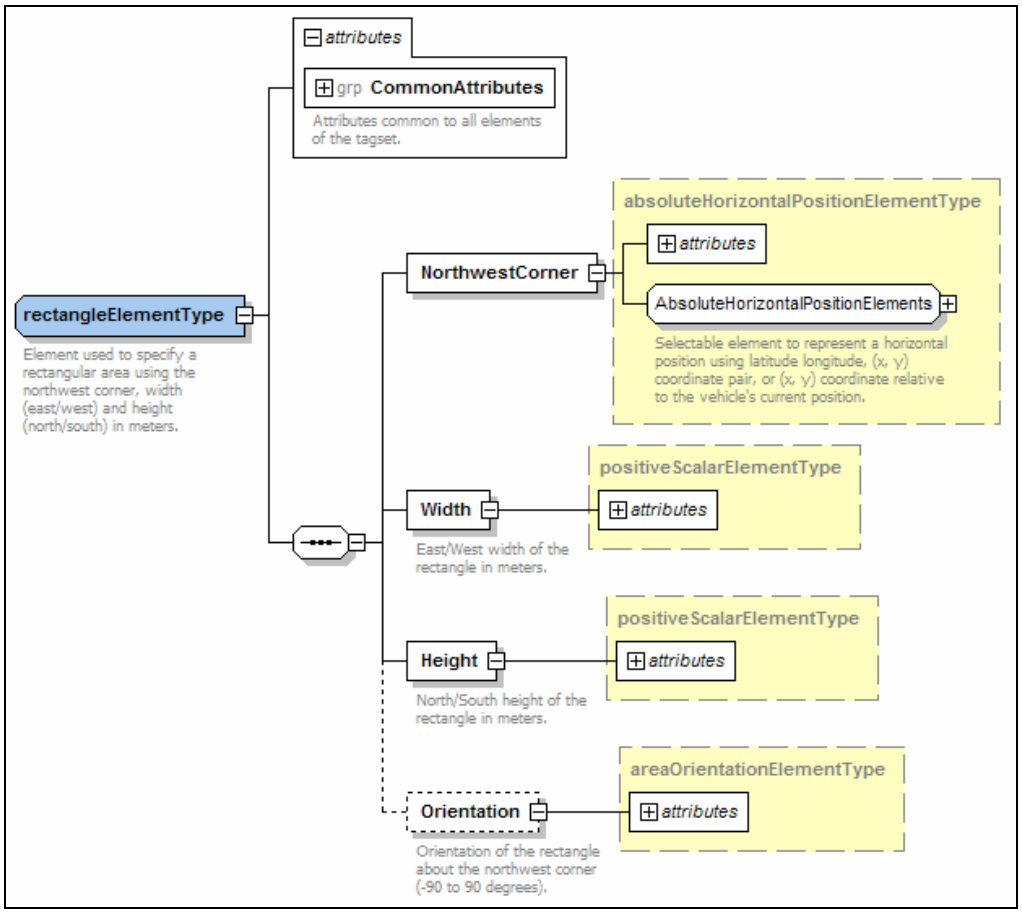


Figure A.4. The AVCL rectangleElementType for Specifying a Geographic Area as a Rectangle

Also useful in various portions of an autonomous vehicle data model is the ability to represent vehicle and contact states. The position elements described above are applicable here, however vehicle orientation and velocity are also important. AVCL

provides four basic types for this purpose: orientationElementType, worldCoordinateVelocityElementType, bodyCoordinateVelocityElementType and dopplerVelocityElementType. Each of these consists of childless elements with the attributes described in Tables A.5 through A.7.

Attribute	Use	Type	Description
phi	optional	orientationType	Euler angle rotation about the body or world coordinate frame X axis (bank).
theta	optional	orientationType	Euler angle rotation about the body or world coordinate frame Y axis (pitch).
psi	optional	orientationType	Euler angle rotation about the body or world coordinate frame Z axis (yaw).

Table A.5. AVCL Orientation Element Type Attributes

Attribute	Use	Type	Description
xDot	optional	xsd:double	Linear velocity (meters per second) along the X axis of the world coordinate frame (north positive).
yDot	optional	xsd:double	Linear velocity along the Y axis of the world coordinate frame (east positive).
zDot	optional	xsd:double	Linear velocity along the Z axis of the world coordinate frame (down positive).
phiDot	optional	xsd:double	Angular velocity (degrees per second) about the X axis of the world coordinate frame.
thetaDot	optional	xsd:double	Angular velocity about the Y axis of the world coordinate frame.
psiDot	optional	xsd:double	Angular velocity about the Z axis of the world coordinate frame.
u	optional	xsd:double	Linear velocity along the X axis of the body coordinate frame (vehicle forward).
v	optional	xsd:double	Linear velocity along the Y axis of the body coordinate frame (vehicle right).
w	optional	xsd:double	Linear velocity along the Z axis of the body coordinate frame (vehicle down).
p	optional	xsd:double	Angular velocity about the X axis of the body coordinate frame.
q	optional	xsd:double	Angular velocity about the Y axis of the body coordinate frame.
r	optional	xsd:double	Angular velocity about the Z axis of the body coordinate frame.

Table A.6. Attributes Defined by the AVCL Complex Types for Representing Velocity Relative to the World-Fixed and Body-Fixed Coordinate Frames

Attribute	Use	Type	Description
speedOverGroundU	optional	xsd:double	Computed or measured forward linear velocity over the ground.
speedOverGroundV	optional	xsd:double	Computed or measured lateral linear velocity over the ground.
speedThroughMediumU	optional	xsd:double	Computed or measured forward linear velocity through the air or water.
speedThroughMediumV	optional	xsd:double	Computed or measured lateral linear velocity through the air or water.

Table A.7. AVCL Attributes for Representing Doppler-Based Velocity Over the Ground and Through the Air or Water

Similar in concept to AVCL's state types are a set of groups used to specify depth, altitude, and speed. These elements are used to indicate partial vehicle or contact state, ordered behavior requirements, and to define the vertical characteristics of areas. Each group provides for the selection of a single element. The names and types of the elements associated with each of these groups are described in Table A.8. In some instances, these types are further grouped using the AVCL VerticalBlockElements group that provides for the selection of a single subgroup from the table.

Type	Element Name	Type	Description
DepthTypeElements	Depth	nonNegativeScalarElementType	Depth (meters) of a vehicle or contact below the ocean surface.
	Altitude	nonNegativeScalarElementType	Altitude (meters) of a vehicle or contact above the ocean surface.
DepthBlockElements	DepthBlock	numericalBlockElementType	Specifies a depth below the surface range.
	AltitudeBlock	numericalBlockElementType	Specifies an altitude above the bottom range.
	DepthAltitudeBlock	numericalBlockElementType	Specifies a minimum depth to minimum altitude above the bottom range.
AltitudeTypeElements	AGLAltitude	nonNegativeScalarElementType	Specifies an above-ground-level altitude (meters).
	MSLAltitude	nonNegativeScalarElementType	Specifies a mean-sea-level altitude (meters).
AltitudeBlockElements	AGLAltitudeBlock	numericalBlockElementType	Specifies a range of above ground level altitudes.
	MSLAltitudeBlock	numericalBlockElementType	Specifies a range of mean sea level altitudes.
	AGLMSLALtitudeBlock	numericalBlockElementType	Specifies a maximum mean sea level to minimum above ground level altitude range.
SpeedTypeElements	Speed	nonNegativeScalarElementType	Specifies a speed in meters per second.
	Knots	nonNegativeScalarElementType	Specifies a speed in nautical miles per hour (knots).

Table A.8. AVCL Groups Used to Specify Depth, Altitude and Speed

The preceding types are among the most commonly utilized. Other complex types defined by the AVCL schema are less widely used. These are discussed in conjunction with the sections of AVCL to which they are applicable.

D. TOP-LEVEL DOCUMENT STRUCTURE

There are three valid AVCL root element tags: "AVCL," "AVCLMessage" and "AVCLMessageList." A document with an "AVCL" root element defines a single mission definition or contains results from an executed mission. A document with an "AVCLMessage" root element contains a single inter-vehicle message. Finally, documents with an "AVCLMessageList" root element contain one or more messages.

The high-level structure of the "AVCL" element is depicted in Figure A.5. Children include an optional "head" element containing "meta" elements that can encode arbitrary descriptive information and a required "body" element that contains the actual mission definition and results. Attributes associated with these and other immediate descendants of the "AVCL" element are listed in Table A.9.

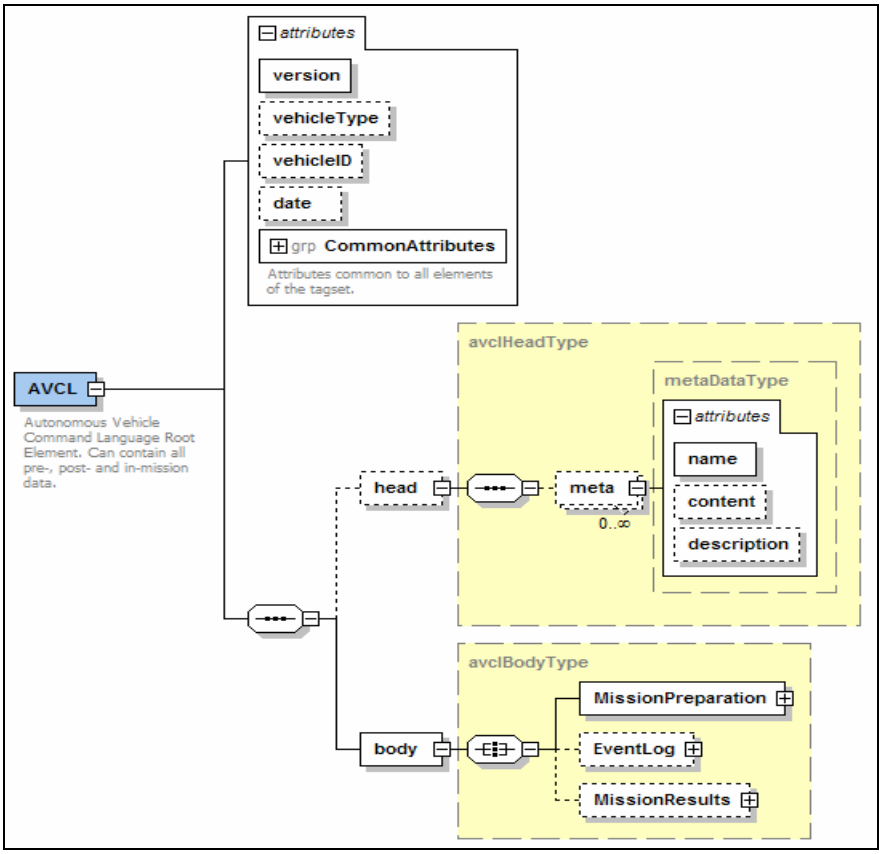


Figure A.5. Structure of the AVCL Root Element for Mission Definition and Mission Results Documents

The “body” element contains a required “MissionPreparation” element and optional “EventLog” and “MissionResults” elements (the latter two are discussed in Section F). The “MissionPreparation” element (Figure A.6) has one required child (“UnitsOfMeasure”) which indicates the units used throughout the document. Optional children consist of a “GeoOrigin” element defining the origin of the world-fixed coordinate frame, a “Configuration” element describing the characteristics of the vehicle for which the document is intended, and a mission-definition element (discussed in Section E). The “Configuration” element contains optional “Dimensions,” “OperatingCharacteristics,” and “DegreesOfControl” elements, and zero or more “Capability” elements. The attributes of the “Dimensions” and “DegreesOfControl” elements are listed in Table A.9 and the “Capability” element has a single “value” attribute constrained by one of the previously discussed vehicle-specific capability types (e.g., uuvCapabilityType). Attributes associated with an “OperatingCharacteristics” element vary with vehicle type (see the AVCL schema for a further description).

The remaining AVCL root elements, “AVCLMessage” and “AVCLMessageList” are used to encode inter-vehicle message data. Their content models are discussed in detail Section G of this appendix. The high-level structure of both elements (depicted in Figure A.7) is similar to that of the “AVCL” element in that the immediate children of both are a “head” element (optional for the “AVCLMessageList” element) and a “body” element. In fact, the content model of the “head” child of an “AVCLMessageList” element is identical to that of an “AVCL” element as are all of the root-element attributes. The “body” child element of an “AVCLMessageList” element contains one or more “AVCLMessage” elements. The attributes of an “AVCLMessage” element include the common AVCL attributes and a mandatory “version” attribute with a fixed value of “1.0.” The content models of the “head” and “body” elements of an “AVCLMessage” element are discussed in Section G.

Element	Attribute	Use	Type	Description
AVCL	version	required	fixed "1.0"	Identifies the AVCL version.
	vehicleType	optional	xsd:string	Identifies a vehicle type to which the document applies.
	vehicleID	optional	xsd:unsignedShort	Identifies a specific vehicle to which the document applies.
	date	optional	xsd:date	Document or mission date.
meta	name	required	xsd:string	The type of annotating information encoded.
	content	optional	xsd:string	Amplifying information.
UnitsOfMeasure	distance	required	fixed "meters"	All AVCL distances are specified in meters.
	angle	required	fixed "degrees"	AVCL angles are degrees.
	mass	required	fixed "kilograms"	AVCL masses are kilograms.
	time	required	fixed "seconds"	AVCL times are seconds.
GeoOrigin	latitude	required	latitudeType	Latitude of the world-fixed coordinate frame origin.
	longitude	required	longitudeType	Longitude of the world-fixed coordinate frame origin.
	radiusOfInterest	optional	positiveScalarType	Rough estimate of the operating area size.
Dimensions	length	required	positiveScalarType	Vehicle longitudinal length.
	width	required	positiveScalarType	Vehicle lateral width or wingspan.
	height	required	positiveScalarType	Vehicle vertical height.
	mass	required	positiveScalarType	Vehicle mass.
DegreesOfControl	longitudinal	optional	xsd:boolean	Indicates fore / aft controllability (default true).
	lateral	optional	xsd:boolean	Indicates left / right controllability (default false).
	vertical	optional	xsd:boolean	Indicates vertically controllability (default false).
	roll	optional	xsd:boolean	Indicates roll controllability (default false).
	pitch	optional	xsd:boolean	Indicates pitch controllability (default false).
	yaw	optional	xsd:boolean	Indicates yaw controllability (default true).

Table A.9. AVCL Attributes Associated with the "AVCL" Element and its Immediate Descendants

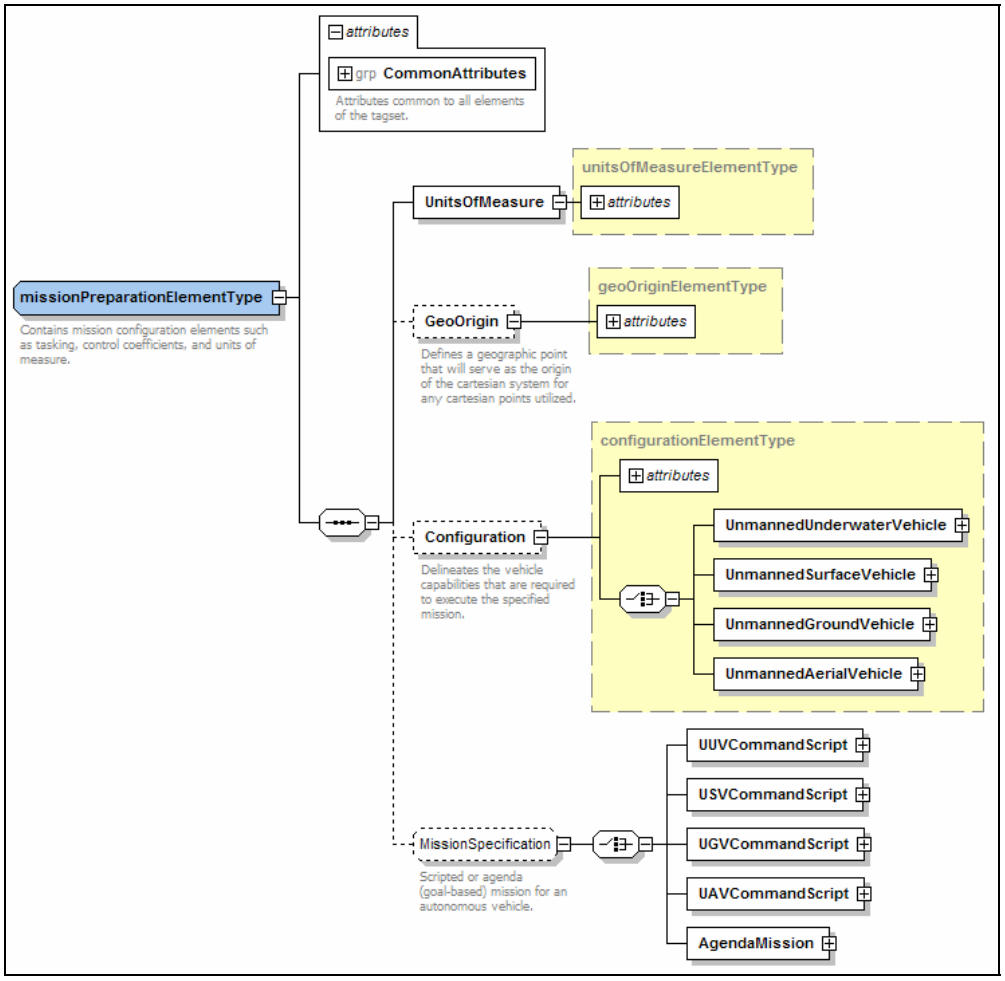


Figure A.6. The Content Model of the AVCL "MissionPreparation" Element

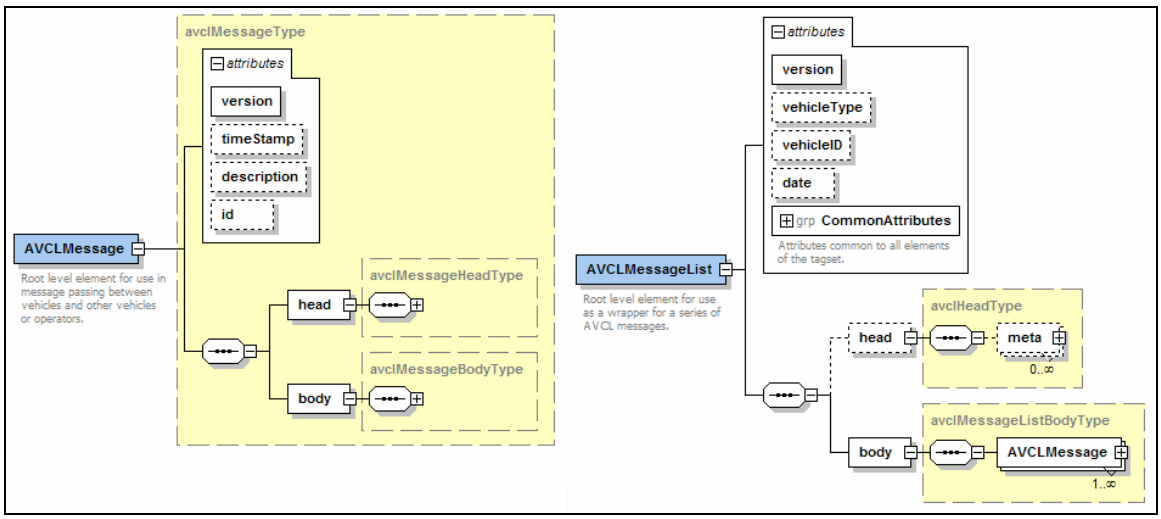


Figure A.7. AVCL Document Root Elements for Inter-Vehicle Message Passing

E. MISSION DEFINITION

1. Task-Level Behavior Scripts

a. Overview

AVCL provides two basic methods for specifying a mission. The first is using task-level behavior scripts. Depicted in Figure A.6, a task-level behavior script is specified using one of the vehicle-type-specific script tags from the MissionSpecification group. Each of these elements contains a sequence of one or more task-level behaviors associated with the particular vehicle type. Available behaviors for each vehicle type are described in the following sections.

b. UUV Behaviors

AVCL defines 30 UUV task-level behaviors. Among these are behaviors classifiable as closed-loop / terminating, closed-loop / open-ended, open-loop, and miscellaneous behaviors as described in Chapter IV as well as a number of behaviors that pertain to missions run in simulation. The remainder of this section describes these behaviors, each of which is specified with an element with the behavior name.

The CompositeWaypoint behavior (Figure A.8) is used to succinctly define a predefined pattern of waypoints. The "CompositeWaypoint" element's first child element is used to specify the type of pattern (IMO and ICAO, 98) using one of the elements depicted in Figure A.9. The required attributes (described in Table A.10) parametrically specify the characteristics of the pattern and the child element specifies the position of the first waypoint. Depth below the surface or altitude above the bottom that the vehicle is to maintain throughout the pattern is included with a "Depth" or "Altitude" element. An optional "HomingMode" element determines whether the vehicle is to adhere to the tracks defined by the waypoints or proceed directly to each waypoint. Vehicle speed is optionally ordered with the next child element (content models are covered with the same-name behaviors). Finally, optional "GpsFixes," "Standoff," and "TimeOut" elements are used to specify the number of fixes to obtain over the course of the pattern, how close (meters) the vehicle must get to each waypoint before proceeding to the next, and how long (seconds) the vehicle has to achieve each waypoint.

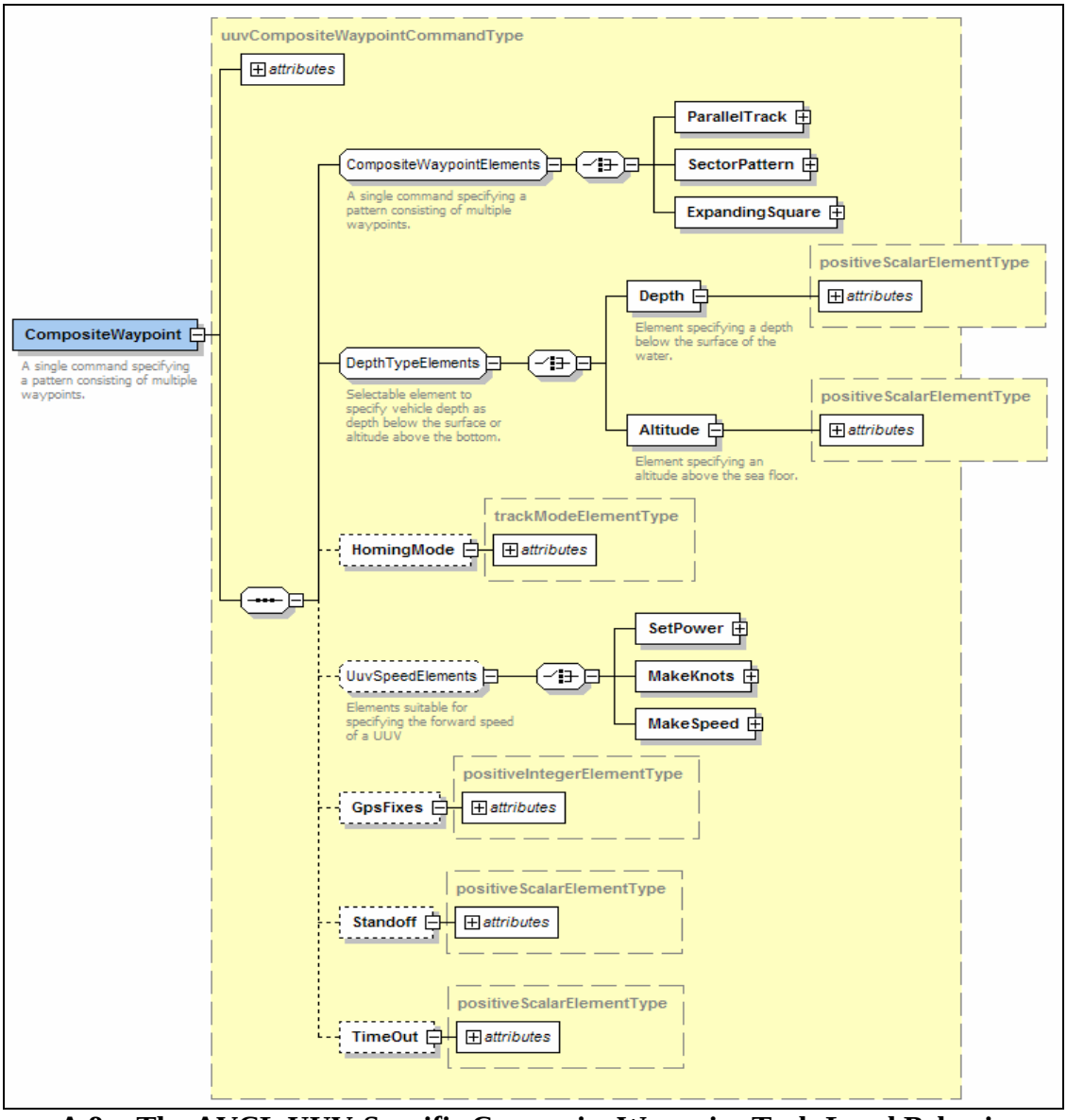


Figure A.8. The AVCL UUV-Specific Composite Waypoint Task-Level Behavior

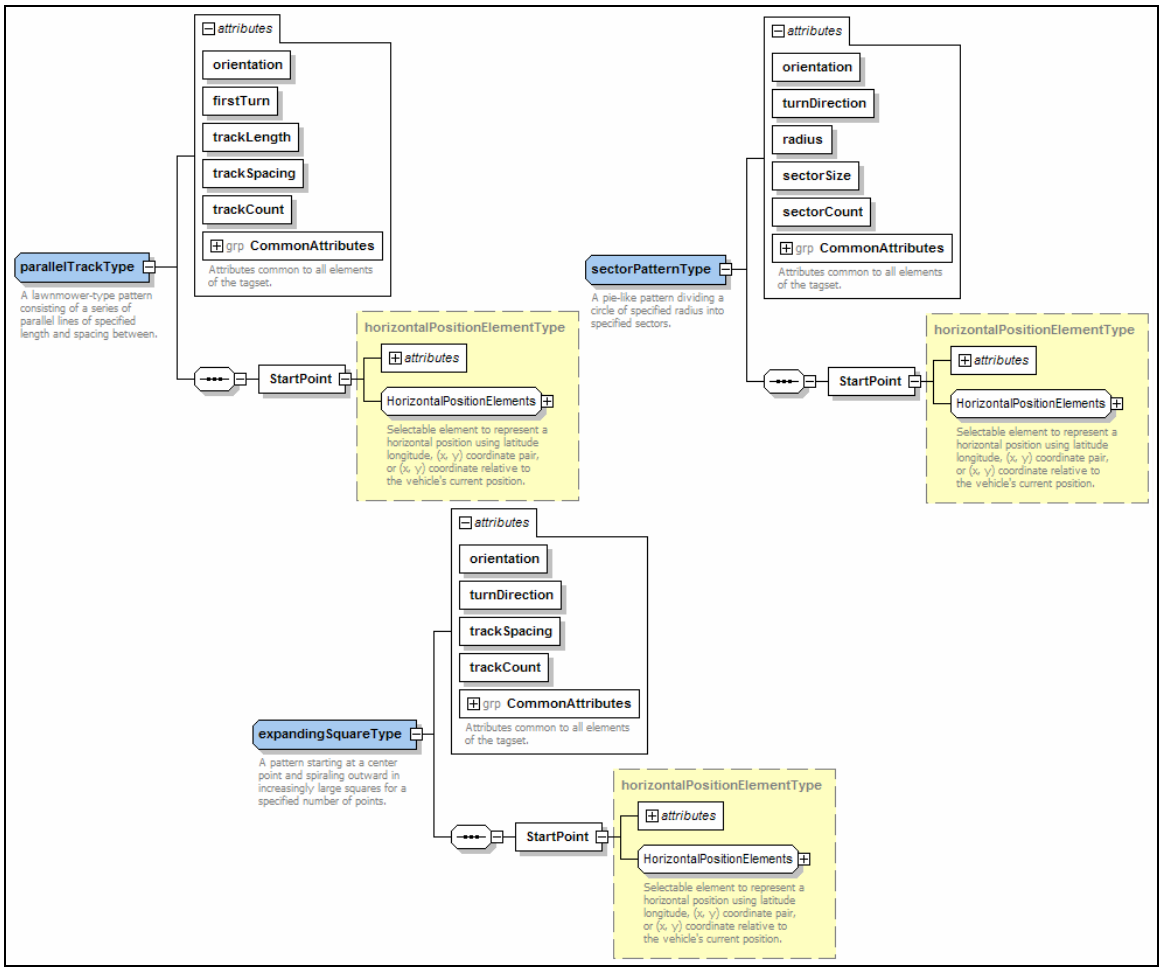


Figure A.9. AVCL Elements for Parametrically Specifying a Pre-Defined Waypoint Pattern

Attribute	Type	Description
orientation	headingType	Heading of the first leg of the pattern.
firstTurn	turnDirectionType	Direction of the pattern's first turn.
trackLength	positiveScalarType	Length (meters) of a parallel track pattern's tracks.
trackSpacing	positiveScalarType	Distance (meters) between the pattern's parallel tracks.
trackCount	positiveIntType	The number of tracks or legs in a parallel-track or expanding-square pattern.
sectorCount	positiveIntType	The number of sectors in a sector pattern.
sectorWidth	positiveScalarType	Width (degrees) of the sectors in a sector pattern
radius	positiveScalarType	The radius (meters) of the enclosing circle of a sector pattern.

Table A.10. Attributes Associated with AVCL Composite Waypoint Elements

A FollowBeacon behavior is used to order the vehicle to proceed directly towards source of a beacon or transponder. The specific beacon is optionally defined with a "Beacon" element of type stringElementType (if no beacon is listed, vehicle defaults determine the nature of the beacon). An optional "TimeOut" element is used to specify how long the vehicle is to proceed towards the beacon. If no time out is included or the vehicle reaches the source before the time out expiration, the behavior terminates.

A GpsFix behavior directs the vehicle to the surface for a GPS fix (other control aspects are not affected). It is defined using an extension to the booleanElementType. A "value" attribute of "true" initiates the behavior, while a value of "false" terminates an active GpsFix behavior (a GpsFix behavior with a "value" attribute of "false" is ignored if no GpsFix behavior is active). An optional "timeOut" attribute (nonNegativeScalarType) is used to set the maximum allowable time the behavior is to be active. Omitting the "timeOut" attribute or setting it to zero means that the vehicle is to remain surfaced until the fix is obtained. The GpsFix behavior terminates when a fix has been obtained, the behavior times out, or a new GpsFix behavior with a "value" attribute of "false" is activated. Upon termination, the previously active depth-control behavior resumes.

A Hover behavior is used to direct the vehicle to maintain a fixed position at a specific location (i.e., hover in place). Defined as depicted in Figure A.10, behavior requirements are encoded in the children (all optional) of a "Hover" element. The first child is from the HorizontalPositionElements group and specifies the hover location in either absolute terms or relative to the vehicle's position upon behavior activation. If this element is omitted, the vehicle is to hover at the current location. The next element, from the DepthTypeElements group, specifies depth below the surface or altitude above the bottom for both transit and hover. Omitting this element retains the previous depth behavior. Inclusion of a "Heading" element (headingElementType) specifies the heading to maintain while hovering, while omission retains the previous heading behavior in the hover. An optional "ObtainGps" element (booleanElementType) can direct the vehicle to obtain a GPS fix during while transiting to the hover point. A "Standoff" element (positiveScalarTypeElement) defines the acceptable distance (meters) from the ordered hover location. Finally, a "TimeOut" element (positiveScalarTypeElement) specifies the

maximum time allotted to reach the designated location and establish a steady hover (omission of this element means that the behavior will not time out). The Hover behavior terminates when a steady is established or the behavior times out unless extended with a Wait or WaitUntilTime behaviors (description to follow).

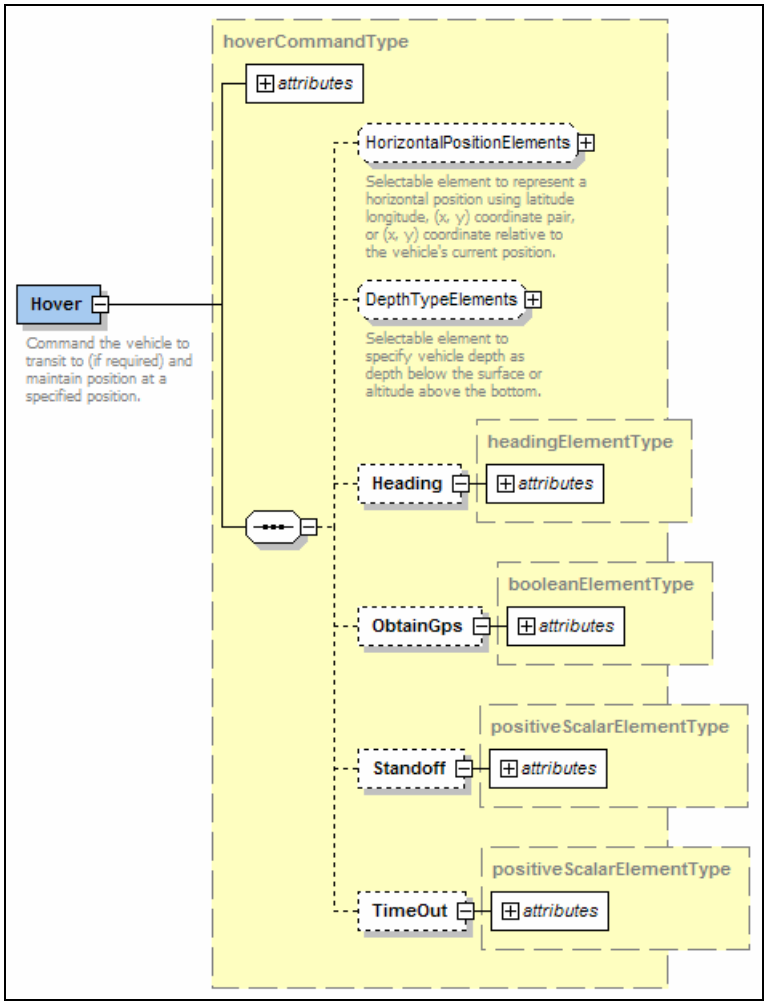


Figure A.10. An AVCL Element for Initiating a UUV Hover Behavior

Similar to the Hover behavior, a Loiter behavior also directs the vehicle to remain at a specific location, however it does not require the vehicle to remain stationary. Constructed as depicted in Figure A.11, the Loiter behavior parameters are contained in the children of a "Loiter" element. The first child element comes from the HorizontalPositionElements group and specifies the loiter location. If this element is omitted, the current position is to be used. The next child, from the DepthTypeElements group, determines the transit depth below the surface or altitude above the bottom. If

omitted, the currently active depth behavior remains in effect. A transit-speed is ordered using a "SetPower," "MakeKnots," or "MakeSpeed" element with a content model matching that of the behavior of the same name. If no speed is included, the greater of the minimum acceptable transit speed and the most recently ordered speed is to be used. Vehicle speed upon reaching the loiter point is vehicle-specific. A required "LoiterDepth" positiveScalarElementType element specifies the depth at which the vehicle is to loiter. Finally, a "TimeOut" element (positiveScalarElementType) is used to specify a maximum allowable time to reach the loiter point (if omitted, the behavior will not time out). The Loiter behavior terminates upon time out or upon reaching the loiter point (whichever occurs first) unless the behavior termination is suspended with a Wait or WaitUntilTime behavior.

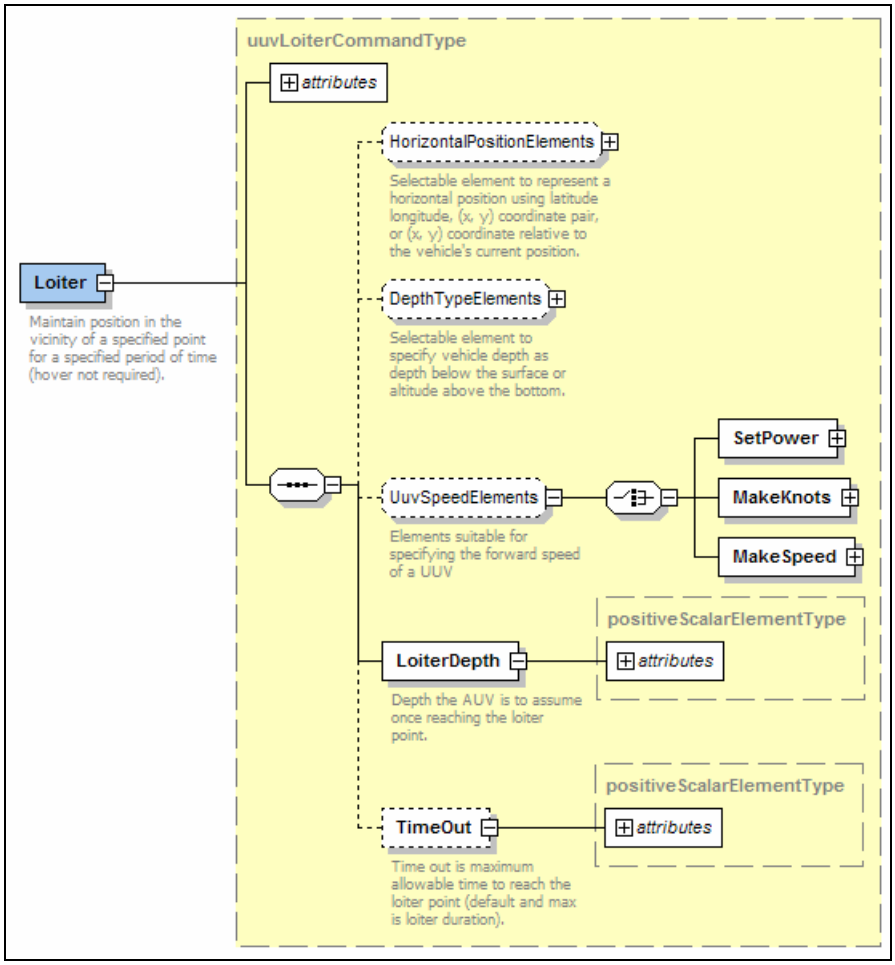


Figure A.11. AVCL Element for Initiating a UUV Loiter Behavior

Available closed-loop / open-ended behaviors include a MakeAltitude, MakeDepth, MakeHeading, MakeKnots, and MakeSpeed. These behaviors are summarized in Table A.11. MakeAltitude and MakeDepth behaviors are defined using a positiveScalarElementType to command an altitude above the bottom or depth below the surface in meters. MakeHeading orders vehicle heading in degrees. MakeKnots and MakeSpeed are used to order a specific vehicle speed in nautical miles per hour (knots) or meters per second respectively. Both behaviors are defined with an extended positiveScalarElementType element with an optional Boolean parameter ("speedOverGround") that determines whether the behavior is ordering speed over the ground or through the water. All of these behaviors remain active until superseded.

Behavior	AVCL Type	Description
MakeAltitude	positiveScalarElementType	Orders the vehicle to maintain a specified altitude above the bottom (meters).
MakeDepth	positiveScalarElementType	Orders the vehicle to maintain a specified depth below the surface (meters).
MakeHeading	headingElementType	Orders the vehicle to maintain a specified heading (degrees).
MakeKnots	extended positiveScalarElementType	Orders the vehicle to maintain a specified speed over the ground or through the water (knots).
MakeSpeed	extended positiveScalarElementType	Orders the vehicle to maintain a specified speed over the ground or through the water (meters per second).

Table A.11. Available AVCL UUV Closed-Loop / Open-Ended Behaviors

As described in Chapter V, the MetaCommand behavior does not directly effect vehicle control, but it provides a useful container for annotating information that effects how the script is interpreted or converted to vehicle-specific formats. The "MetaCommand" element content model matches that of the "meta" element discussed in the previous section in both content and interpretation.

AVCL provides two behaviors for replacing or adding to an executing script. The MissionScript and MissionScriptInline behaviors are both defined using a stringElementType element where the "value" attribute indicates the path to the new task-level behavior script. The MissionScript behavior is used to replace the currently executing script completely. The MissionScriptInline behavior, on the other hand, inserts the newly loaded script into the currently executing script at the current position.

The MoveLateral and MoveRotate behaviors apply only to UUVs with cross-body thrusters. The MoveLateral behavior requires the vehicle to use its thrusters to slide laterally while the MoveRotate behavior requires the vehicle to use its thrusters to rotate about the body-fixed coordinate system Z axis. Both open-loop behaviors are defined with a signedPercentElementType element where the "value" attribute indicates the percentage of available power to apply to the thrusters and both remain active until superseded by a behavior requiring heading, hover, or waypoint control.

A Quit behavior is used to indicate the end of a script. Following activation, all other active behaviors are terminated and the vehicle will initiate any required mission conclusion procedures. The "Quit" element has up to one optional child element (noValueElementType) that is used to indicate the reason the behavior was activated. Allowable child element names are "NormalExit," "MissionAbort," "SystemAbort," and "RecallAbort."

The Realtime behavior is the first that pertains exclusively to missions run in simulation. This behavior is initiated using a booleanElementType element and is used to toggle a simulation between real-time execution (i.e., simulation elapsed time equals elapsed clock time) and fastest possible execution (i.e., simulation elapsed time greater than elapsed clock time). The most recently activated real-time setting remains in effect until superseded.

A SendMessage behavior is used to conduct inter-vehicle communications. The sole child element of a "SendMessage" element is an "AVCLMessage" element that is to be transmitted. The message content model of the is discussed in Section G.

AVCL provides open-loop behaviors to control the rudder, horizontal control planes, propellers and cross-body thrusters. The simplest of these is the SetRudder behavior which uses a signedPercentElementType element to order rudder deflection as a percentage of the maximum authority. On vehicles with bow and stern rudders, the command is applied to the stern rudder and the bow rudder receives the opposite order. As with all open-loop behaviors, the order directs a control setting, but not necessarily a direction of vehicle movement. For instance a positive rudder setting

orders the forward edge of the stern rudder to the right but has the effect of yawing the vehicle to the left. The SetPlanes, SetPower, and SetThruster behaviors are described in Table A.12. The element used to define each of these behaviors contains one child element (signedPercentageElementType) from those listed in the table that identifies the specific controller that the behavior orders. As with the SetRudder behavior, the control setting is ordered as a percentage or maximum authority. Unless otherwise specified, behaviors that attempt to affect controllers not possessed by the target vehicle are ignored. All of the open-loop behaviors remain in effect until superseded.

Behavior	Child Element	Order Description
SetPlanes	BowPlane	Percent of maximum deflection for the bow horizontal planes (positive indicates forward edge up).
	SternPlane	Percent of maximum deflection for the aft horizontal control planes (positive indicates forward edge up)
	AllPlanes	Combined order for fore and aft control planes. Planes will deflect in opposite directions. The sign of the "value" attribute is applied to the stern plane.
SetPower	PortPropeller	Percent of maximum power from the port propeller (if single-prop, the order applies to the single propeller).
	StarboardPropeller	Percent of maximum power from the starboard propeller If single-prop, the order applies to the single propeller).
	CenterlinePropeller	Percent of maximum power from the port propeller (if single-prop, the order applies to the single propeller).
	AllPropellers	Orders a percentage of maximum power from all propellers.
SetThruster	BowLateralThruster	Percent of maximum power from the bow-lateral cross-body thruster (positive pushes the vehicle nose to the right)
	SternLateralThruster	Percent of maximum power from the stern-lateral cross-body thruster (positive pushes the vehicle stern to the right).
	LateralThrusters	Percent of maximum power from all lateral cross-body thrusters (positive pushes the vehicle to the right).
	BowVerticalThruster	Percent of maximum power from the bow-vertical cross-body thruster (positive pushes nose of vehicle down).
	SternVerticalThrusters	Percent of maximum power from all vertical cross-body thrusters (positive pushes the vehicle stern down).
	VerticalThrusters	Percent of maximum power from the stern-vertical cross-body thruster (positive pushes stern of vehicle down).

Table A.12. AVCL Open-Loop Behaviors for Control of Propellers, Cross-Body Thrusters and Horizontal Control Planes

The SetPosition behavior is provided as a means of resetting the vehicle's internally maintained position data to a new value. The "SetPosition" element has a mandatory child element from the AbsoluteHorizontalPositionElements group that defines the new geographic position and an optional "Depth" element (positiveScalarElementType) specifying the current depth below the surface.

The SetStandoff behavior is used to change the capture radius of waypoints, hover points, and loiter points (i.e., the distance at which a waypoint, loiter point, or hover point is considered achieved). Defined with a positiveScalarElementType element, the "value" attribute specifies the new capture radius in meters. While not directly affecting vehicle control, the standoff distance is an implicit part of all Waypoint, Hover, and Loiter behaviors and remains in effect until superseded.

Used primarily in simulations, the SetTime behavior changes the vehicle's internally maintained time. It is defined using a positiveScalarElementType element where the "value" attribute specifies the number of seconds since beginning execution.

Also used primarily in simulations, the SetTimeStep behavior changes the amount of time between each iteration of the main control loop. Also defined using a positiveScalarElementType element, the "value" attribute determines the time (seconds) for each loop iteration. The new closed-loop time step remains in effect until superseded.

The Trace behavior is the final UUV task-level behavior intended for simulations. It is used to enable and disable any verbose output based on the "value" attribute of a booleanElementType element. The setting specified by a Trace behavior remains in effect until superseded.

The Wait and WaitUntilTime behaviors do not directly influence vehicle control, but are used to extend the activation period of currently active behaviors, most commonly, Hover or Loiter. Both are defined with positiveScalarElementType elements. The "value" attribute of a "Wait" element defines the amount of time (seconds) following activation that the behavior is to remain active. The "value" attribute of a "WaitUntilTime" element, on the other hand, defines an absolute clock time (seconds from execution start) at which the behavior is to deactivate. While active, a Wait or WaitUntilTime behavior will preclude the activation of further behaviors. Under normal

circumstances, a Wait or WaitUntilTime behavior will remain active until its termination time unless terminated by a higher-level in a multi-layer control architecture.

The final UUV task-level behavior is the Waypoint behavior. It is used to order the vehicle to transit to a specific geographic location and is defined with an element of the form of Figure A.12. The waypoint position is specified in relative or absolute terms with an element from the HorizontalPositionElements group. Depth below the surface or altitude above the bottom is optionally ordered using an element from the DepthTypeElements group. An optional trackModeElementType element can order the vehicle to continuously proceed directly towards the waypoint or track along the path between the previous and current waypoints. An en route GPS fix is ordered with an optional "ObtainGps" element (booleanElementType). Finally, the waypoint capture radius is optionally defined with a "Standoff" element and the time allotted to reach the waypoint is optionally specified with a "TimeOut" element (both positiveScalarElementType). A Waypoint behavior terminates upon reaching the waypoint or upon timing out (the behavior will not time out if the "TimeOut" element is omitted) unless its activation is extended using a Wait or WaitUntilTime behavior.

c. UGV Behaviors

The majority of task-level behaviors available for UGV use are identical to those for UUV use. The FollowBeacon, MakeHeading, MakeKnots, MakeSpeed, MetaCommand, MissionScript, MissionScriptInline, Quit, Realtime, SendMessage, SetStandoff, SetTime, SetTimeStep, Trace, Wait, and WaitUntilTime behaviors all fall into this category. The remaining UGV behaviors also have UUV counterparts, but their content models vary somewhat because of the inherent differences in vehicle capabilities.

All other UGV behaviors also have UUV counterparts, but their content models are not identical. The CompositeWaypoint, Loiter, SetPosition, and Waypoint behaviors make up this category. These behaviors are identical to their UUV counterparts in execution, semantics, and activation / termination criteria. The content differences consist of the absence of elements and attributes relating to depth below the surface or altitude above the bottom, and the elimination percentage of maximum available power as a means of specifying speed (UGV speed is specified with "MakeKnots" and "MakeSpeed" elements).

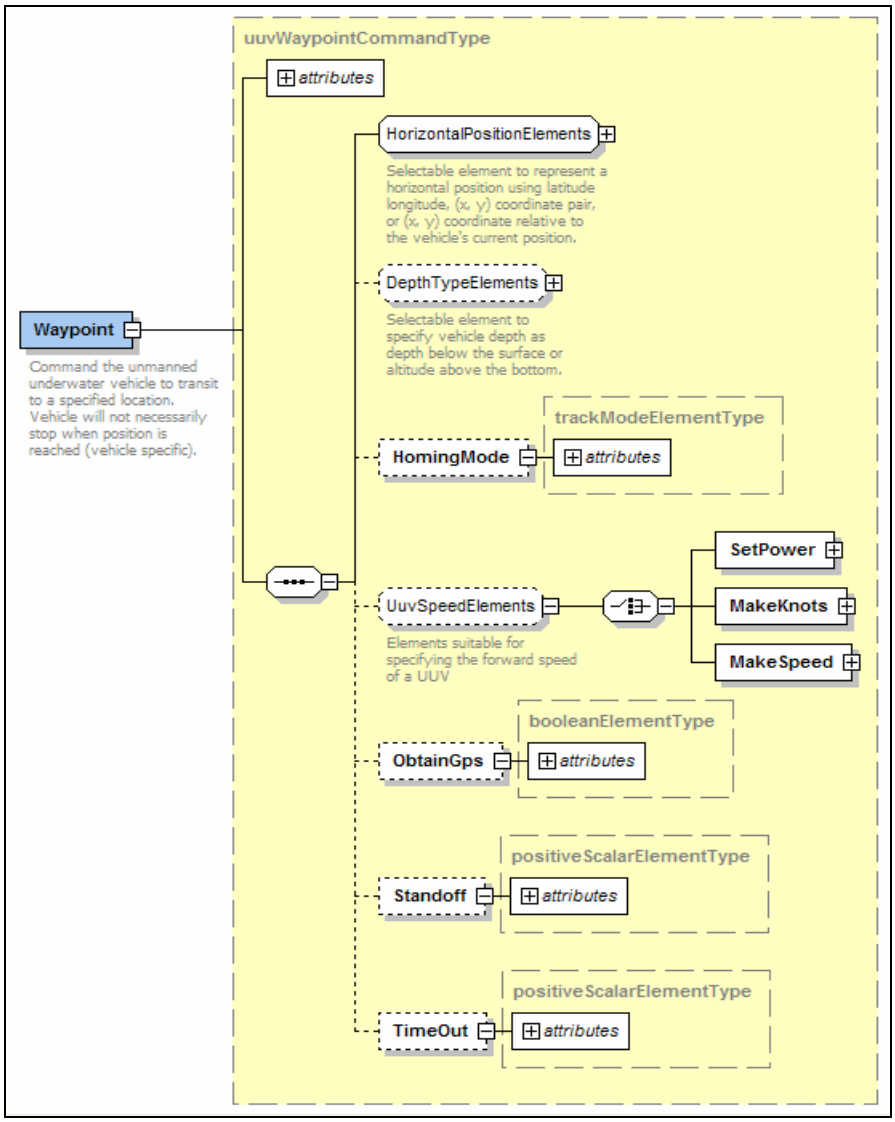


Figure A.12. An AVCL Element for Initiating a UUV Waypoint Behavior

d. USV Behaviors

The elements used to specify UGV behaviors are all available for use in specifying the same behavior for a USV without modification. Additionally, the elements used to specify the UUV SetPower and SetRudder behaviors are used to specify the USV behavior of the same name. In all cases, the content model and semantics of the element used to define the USV behavior are identical to those of the UGV or UUV behavior as are the behavior activation and termination criteria.

e. UAV Behaviors

Not surprisingly, the content models and semantics of the majority of elements for specifying UAV task-level behaviors match those of the corresponding behaviors for one or more other vehicle types. The definition elements of the UAV FollowBeacon, MakeHeading, MakeKnots, MakeSpeed, MetaCommand, MissionScript, MissionScriptInline, Quit, Realtime, SendMessage, SetStandoff, SetTime, SetTimeStep, Trace, Wait, and WaitUntilTime behaviors are identical to those of all other vehicles. Additionally, definition element for the UAV SetRudder behavior matches those of the corresponding UUV and USV behavior.

The elements for defining the UAV CompositeWaypoint, Loiter, SetPosition, and Waypoint behaviors are similar to those of the other vehicle types as well. The UAV "SetPosition" element, for instance, differs from that of the corresponding UUV element only in the replacement of the optional "Depth" child element with an optional "MSLAltitude" element to specify the vehicles current mean-sea-level altitude. The UAV "CompositeWaypoint" element (Figure A.13) differs from that of the UUV in two regards. The first is an element from the AltitudeTypeElements group (replacing one from the DepthTypeElements group) that specifies the above ground level or mean sea level altitude for pattern's waypoints. The second difference is that as with UGVs and USVs, speed is orderable in knots or meters per second, but not as a percentage of maximum power. The UAV "Loiter" and "Waypoint" elements (Figure A.14) also replace depth and speed elements of the UUV-specific element in this way.

Additionally, AVCL provides a number of closed-loop / open-ended and open-loop behaviors that do not correspond to behaviors of another vehicle type. These are listed and described in Table A.13. As with other closed-loop / open-ended and open-loop behaviors, these UAV behaviors remain active until superseded.

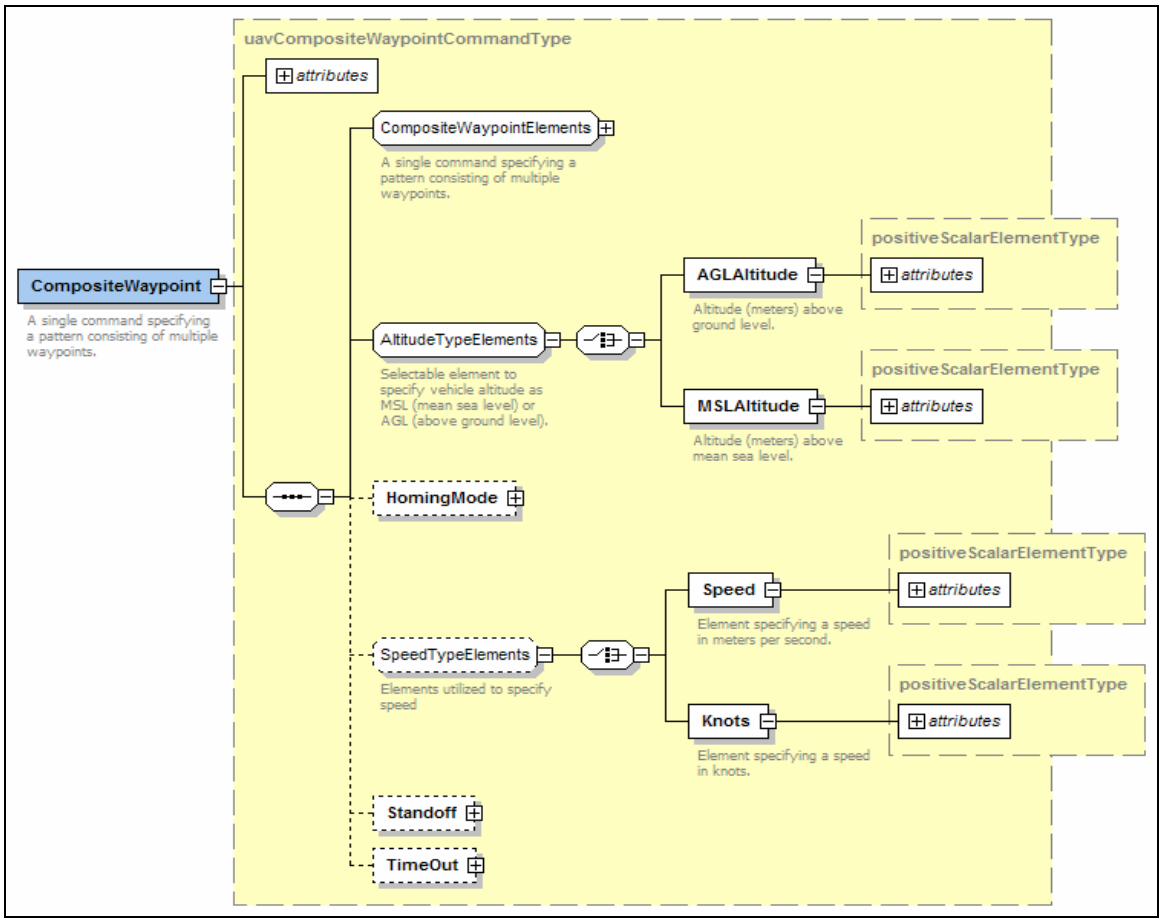


Figure A.13. An AVCL Element for Initiating a UAV Composite Waypoint Behavior

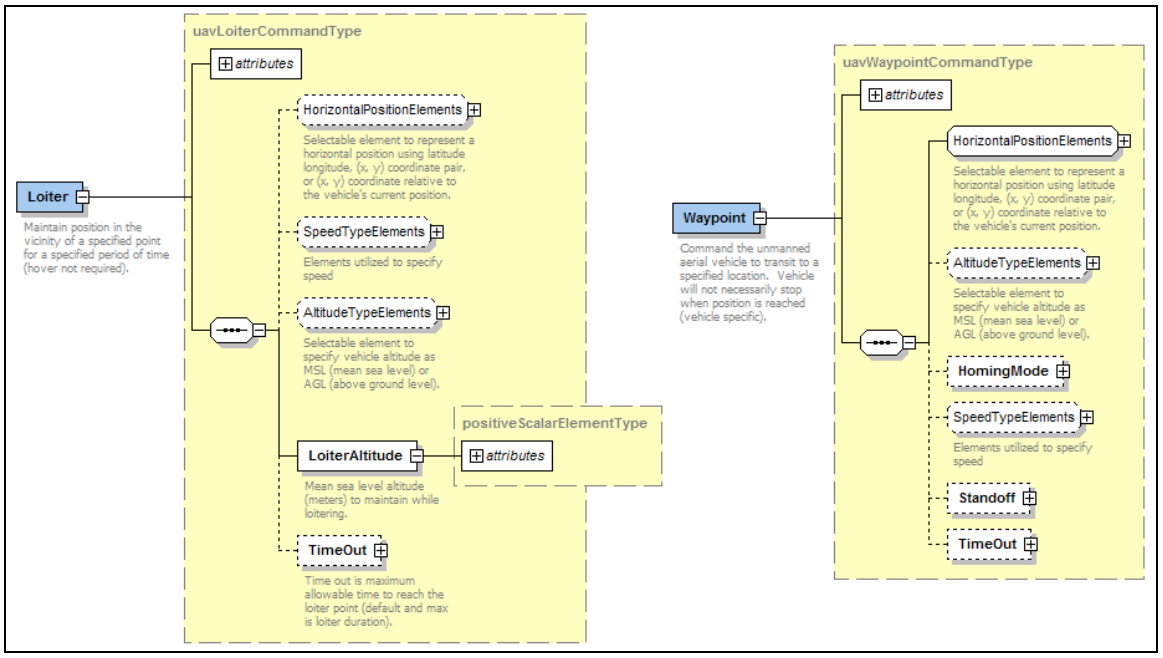


Figure A.14. AVCL Elements for Initiating UAV Loiter and Waypoint Behaviors

UAV Behavior	Element Type	Description
MakeAltitudeAGL	positiveScalarElementType	Orders the vehicle to maintain the specified above ground level altitude (meters).
MakeAltitudeMSL	positiveScalarElementType	Orders the vehicle to maintain the specified mean sea level altitude (meters).
MakeClimbRate	signedPercentElementType	Orders the vehicle to climb or descend at a percentage of its maximum available rate.
MakeTurnRate	signedPercentElementType	Orders the vehicle to turn left or right (negative) at its maximum available rate.
SetAileron	signedPercentElementType	Orders a percentage of maximum aileron deflection (left and right are opposite, positive order induces a right bank).
SetElevator	signedPercentElementType	Orders a percentage of maximum available elevator or horizontal stabilator deflection (positive is leading edge up).
SetPower	percentElementType	Orders a percentage of maximum available engine power (effects all engines equally).

Table A.13. Elements for Specifying UAV-Specific Closed-Loop / Open-Ended and Open-Loop Behaviors

2. Declarative Missions

a. Overview

The second method of defining a mission using AVCL is as a declarative agenda. An agenda consists of high-level goals that are to be accomplished over the course of a mission and constraints that must be observed throughout the mission's execution. Unlike scripted missions, AVCL agendas are largely vehicle-type independent. AVCL agendas can, however, define operating areas with depth or altitude restrictions that make them inherently applicable only to UUVs or UAVs respectively. The "AgendaMission" element defining an AVCL agenda is the last acceptable child element of the "MissionPreparation" element depicted in Figure A.6.

The structure of the "AgendaMission" element is depicted in Figure A.15. The first two child elements are optional "LaunchPosition" and "RecoveryPosition" elements, both from the AbsoluteHorizontalPositionElements group. These elements are used to specify the geographic position of the vehicle at mission commencement and the desired position at the mission conclusion. If the "LaunchPosition" element is omitted, it is assumed that the launch position is irrelevant (i.e., the vehicle will be able to determine its position at launch). If the "RecoveryPosition" element is omitted but the agenda

includes a “LaunchPosition” element, the specified launch position will also be used for the recovery position. If neither element is present, the assumed vehicle response is to conclude the mission immediately following the success or failure of the final goal. It is allowable, however, for the vehicle to return to its run-time determined launch position following completion of the last goal.

The only required child element of the “AgendaMission” element is the “GoalList” element containing definitions for all of the goals that might be executed over the course of the mission. Mission execution begins with the first goal of the mission. The execution order of all subsequent goals is determined by the success and failure of individual goals and the goal definitions themselves. The content models of individual goals is discussed in a subsequent section of this appendix.

The final child of the “AgendaMission” element is the “ConstraintList” element. This element contains three optional child elements as depicted in Figure A.15. “IngressRouting” and “EgressRouting” elements contain routing, altitude, or depth restrictions for the transit to and from the operating area and “AvoidArea” elements contain geographic areas, depths, or altitudes that are to be avoided during the mission. The content model descriptions for the routeElementType and areaElementType.

b. Route and Area Definition

Routing in an AVCL agenda is used only in conjunction with ingress and egress. It provides a means of specifying geographic points that the vehicle is to utilize or altitude or depth restrictions as it transits to and from the operating area. The routeElementType content model is depicted in Figure A.16. It consists of an optional sequence of “LatitudeLongitude” or “XYPosition” elements that define geographic positions that make up the route and an optional element from the VerticalBlockElements group that describes the route’s depth or altitude requirements. It is permissible in AVCL to define the geography of a route without vertical restrictions or vice versa. In fact, it is syntactically valid for a routeElementType element to contain neither geographic points nor vertical requirements, however, this is not useful in practice.

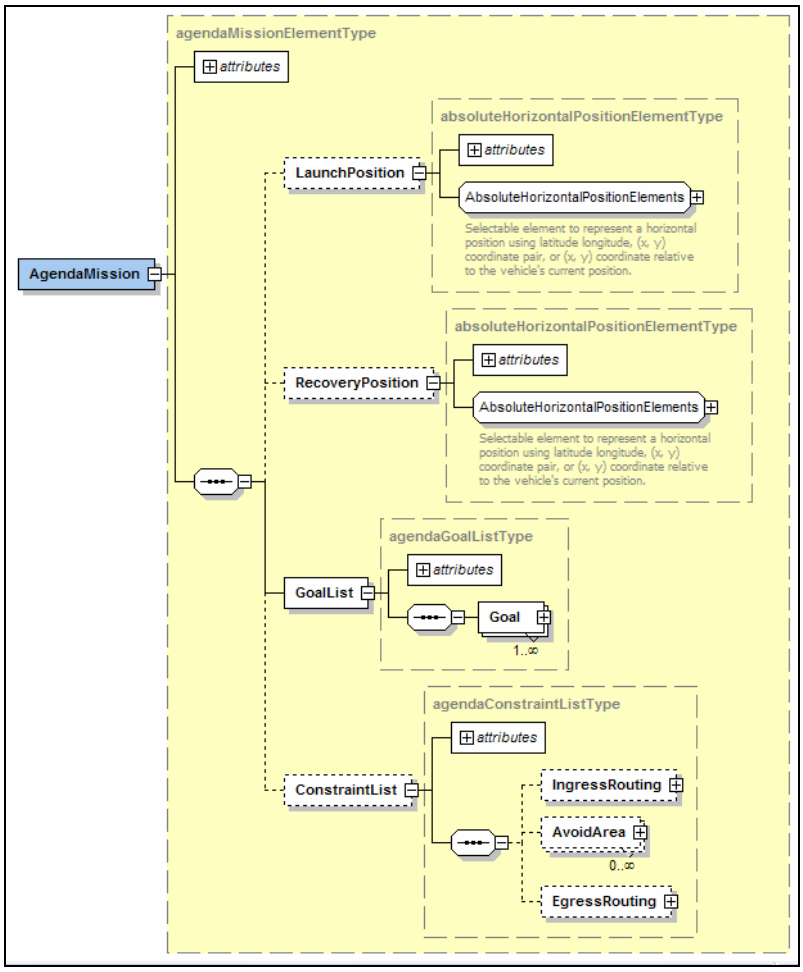


Figure A.15. An AVCL Element for Defining a Declarative Agenda Consisting of High-Level Goals that are to be Accomplished over the Course of a Mission

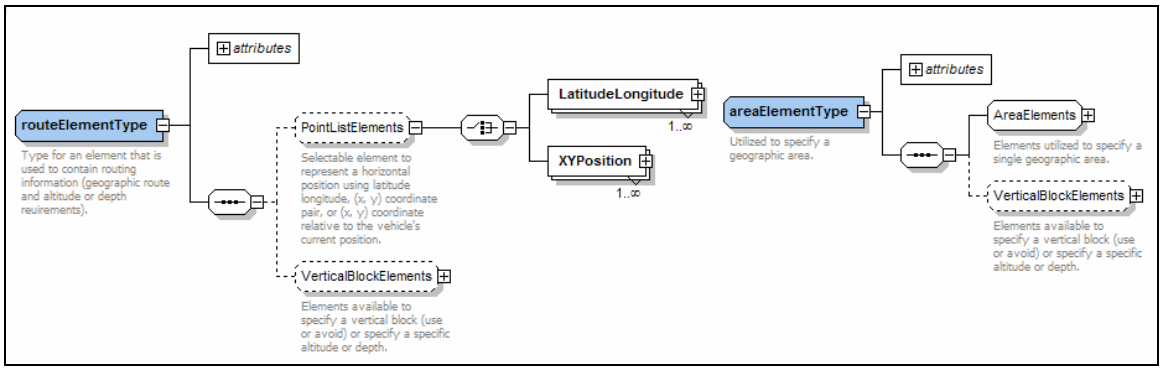


Figure A.16. AVCL Complex Types Used to Define Routes and Areas that can Include Altitude or Depth Components

Also depicted in Figure A.16 is the AVCL areaElementType content model. This element is used to define all avoid areas and operating areas associated with

an AVCL agenda. An areaElementType element has a required child element from the AreaElements group that defines the geographic dimensions of the area. Additionally, the element has an optional element from the verticalBlockElements group that defines the depth or altitude dimensions of the area. If this element is omitted, the area's vertical dimensions are unbounded.

c. Goal Definition

An individual goal of an agenda mission are defined with a "Goal" element of the type depicted in Figure A.17. Among all of the AVCL elements and complex types, the agendaGoalElementType is the only one for which the "id" attribute is required. This allows goals to refer to one another using the "nextOnSucceed" and "nextOnFail" attributes (xsd:IDREF) which are used to define goal sequencing. The goal referred to by the "nextOnSucceed" attribute is to be executed next if the referring goal is completed successfully while the goal referred to by the "nextOnFail" attribute is to be executed next if the goal fails. The remaining attribute that is unique to the agendaGoalElementType is the "alert" element (xsd:boolean). If the "alert" element value is "true," the vehicle is to proceed to the area and wait until directed to commence. If the value is "false," the vehicle is to commence goal execution immediately upon arriving in the operating area.

The first child of a "Goal" element is a member of the GoalElements group and identifies the type of goal and all of the type-specific requirements. The content models of this group will be discussed shortly. The second child element is an areaElementType "OperatingArea" element defining the area in which the goal is to be accomplished. The next child element is either a "Duration" or "Timing" element. A "Duration" element (positiveScalarElementType) is used to specify the amount of time (seconds) the goal has to succeed following commencement. A "Timing" element, on the other hand, has "start" and "stop" attributes (both positiveScalarType) that specify the goal start and end times (seconds from mission commencement). The final child elements are zero or more "ReportingCriteria" elements that specify the conditions under which the vehicle is make reports during goal execution. A "ReportingCriteria" element has a "value" attribute of reportingCriteriaType, and an optional periodicity attribute (positiveScalarType) that is relevant only when the "value" attribute is set to "periodic."

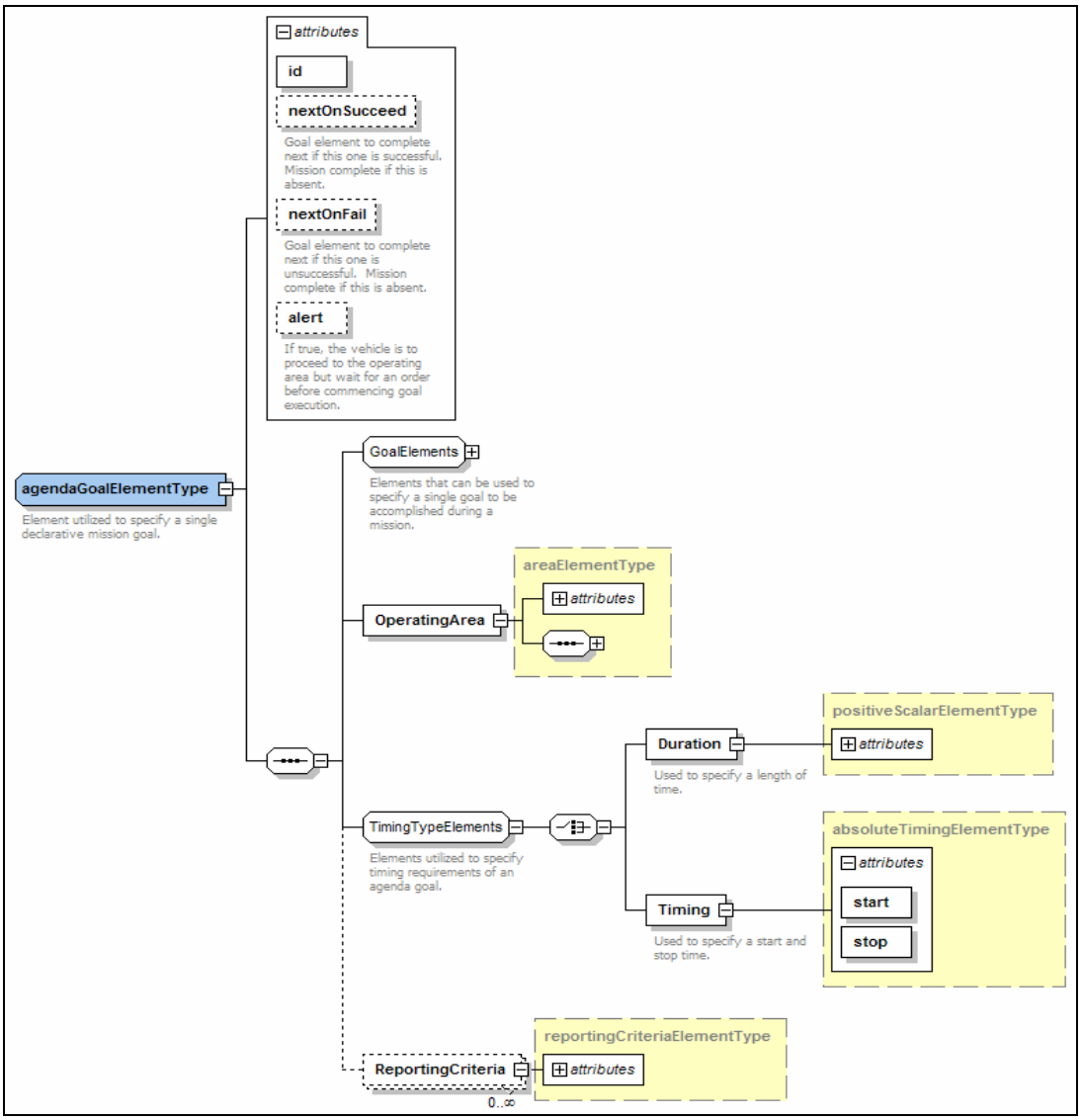


Figure A.17. An AVCL Element for Defining Individual Goals of a Declarative Agenda Mission

AVCL provides for specifying 12 types of goals that are defined using elements (naming matches the goal type): Attack, Decontaminate, Demolish, IlluminateArea, Jam, MarkTarget, MonitorTransmissions, Patrol, Rendezvous, Reposition, SampleEnvironment and Search. The content models of the AVCL elements used for specifying each of these is depicted in Figure A.18, and the attributes associated with each of these elements is described in Table A.14. The following paragraphs describe the semantics of the various goal types. Generally speaking, a goal is considered successful if it is specified as an alert and no execute order is received before the goal

times out, and unsuccessful if the vehicle fails to reach the operating area in time (by the designated start time or the end of the goal duration). Further conditions for goal success or failure are type-specific.

An Attack goal is used to direct the vehicle utilize weapons to engage targets within the operating area. An "Attack" element has zero or more "Target" child elements that identify the target types (omission indicates that vehicle defaults dictate targets). An Attack goal is considered successful if the "singleTarget" attribute has a value of "true" and a target is successfully engaged, or following goal time-out if at least one successful engagement was conducted.

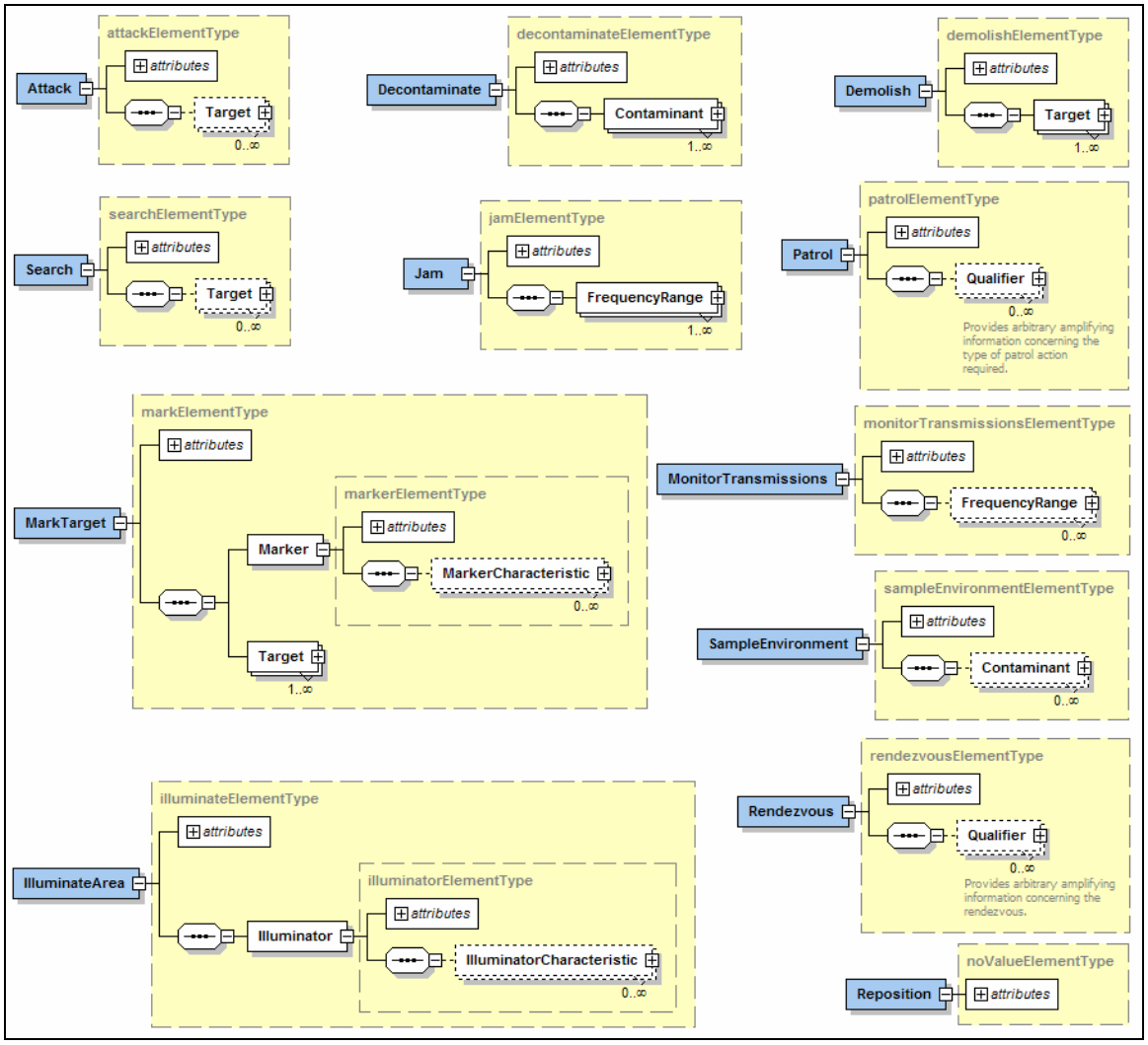


Figure A.18. AVCL Element Content Models for Specifying Goals of a Declarative Agenda Mission

Element	Attribute	Type	Use	Description
Attack	weaponStatus	weaponStatusType	optional (default "tight")	Specifies conditions under which targets can be engaged.
	singleTarget	xsd:boolean	optional (default "false")	Set to "true" if there are potentially multiple targets.
Demolish	singleTarget	xsd:boolean	optional (default "false")	Set to "true" if there are potentially multiple targets.
Rendezvous	targetVehicleID	xsd:string	required	Identifies the vehicle that is to be rendezvoused with.
Search	datumType	datumTypeType	required	Specifies whether the search is to be point or area focused.
	requiredPD	percentType	required	Specifies the required probability of detection of the search.
	singleTarget	xsd:boolean	optional (default "false")	Set to "true" if there are potentially multiple targets.
Target	name	xsd:string	required	Identifies a target type for attack, demolition, or search.
Contaminant	type	contaminantType	required	Identifies a contaminant type.
FrequencyRange	minFrequency	positiveScalarType	required	A minimum frequency of a monitoring or jamming range.
	maxFrequency	positiveScalarType	required	A maximum frequency of a monitoring or jamming range.
	units	frequencyUnitType	optional (default “Hz”)	Wavelength units of the max and min frequencies.
Illuminator	type	illuminatorType	required	Type of light source to use for area illumination.
IlluminatorCharacteristic	value	xsd:string	required	A arbitrary characteristic of an illuminator
Marker	type	markerType	required	Specifies a laser or smoke marker.
MarkerCharacteristic	value	xsd:string	required	Specifies a marker characteristic for a Mark goal.
Qualifier	value	xsd:string	required	Amplifying information about a Patrol or Rendezvous goal.

Table A.14. Attributes of AVCL Elements Used to Define Goals in a Declarative Agenda

A Decontaminate goal directs the vehicle to remove specified contaminants from the operating area. The types of contaminants to be removed are specified using one or more “Contaminant” child elements. A Decontaminate goal is considered successful if the vehicle completes an area-wide environmental-sampling

pattern the entire area and removes any contaminants that are detected. It is considered unsuccessful if it fails to sample the entire area within the allotted time or is unable to remove a detected contaminant.

A Demolish goal is similar to an Attack goal except that it does not contain a "weaponStatus" element. It is used to direct the vehicle to physically destroy one or more targets within the operating area. It is used instead of an Attack goal for targets for which a weapons status is not appropriate (i.e., targets that are not classifiable as "hostile" or "friendly" such as a building).

An IlluminateArea goal directs the vehicle to provide light illumination to an area. The type of illumination is specified with the "Illuminator" child element which call calls for floodlight, spotlight, or pyrotechnic illumination. The goal is considered successful if the vehicle provides illumination over the required time period.

A Jam goal directs the vehicle to prevent the use of radar or communications within the operating area on the frequencies specified by one or more "FrequencyRange" elements. The goal is considered successful if the all targeted emissions during the specified period are jammed.

A MarkTarget goal directs the vehicle to provide a marking for one or more specified targets. A "Marker" child element is used to specify the type of marker that is to be used (laser, smoke, or dye marker) and the "Marker" element can be further qualified with one or more "MarkerCharacteristic" elements. The types of targets to be marked are specified using one or more "Target" elements. A MarkTarget goal is considered successful if at least one target is identified and marked before the goal times out (laser marking requires that the laser remain trained on the target until the goal end time).

A MonitorTransmissions goal directs the vehicle to monitor electronic emissions in specified frequency ranges (communications and / or radar). Ranges to be monitored are specified using zero or more "FrequencyRange" elements with a content model identical to that of the element used in Jam goal definitions. If no frequency ranges are specified, all electronic emissions are to be monitored. A

MonitorTransmissions goal is considered successful if the vehicle is on station with an operational receiver for the prescribed period whether transmissions are detected or not.

A Patrol goal requires the vehicle to wander over the operating area making observations for the specified time. The specific objectives of the goal (e.g., noting vehicular activity, changing environmental characteristics, etc.) are defined using zero or more "Qualifier" elements. A Patrol goal is considered successful if the vehicle is within the operating area with functional mission systems for the required period whether any activity of note is observed or not.

A Rendezvous goal directs the vehicle to proceed to the operating area and make contact with another vehicle (specified with a "targetVehicleID" attribute). Additional rendezvous requirements are defined using zero or more "Qualifier" child elements. A rendezvous goal succeeds only if the target vehicle is contacted.

A Reposition goal causes the vehicle to transition from one geographic area to another. A "Reposition" element contains an optional "Routing" child element complying with the content model of the "IngressRouting" and "EgressRouting" elements. This element defines en route altitude or depth restrictions or geographic points that the vehicle is to use as intermediate waypoints during the transit to the operating area. A Reposition goal is considered successful if the vehicle enters the operating area before the goal times out. If the operating area is specified as a point, the goal will succeed if the vehicle gets within the capture radius.

A SampleEnvironment goal directs the vehicle to test the operating area environment for contaminants or environmental conditions. Contaminants are specified with zero or more "Contaminant" child elements. If no "Contaminant" elements are used, the vehicle is to record the environmental conditions of the area (e.g., temperature, salinity, etc.). A SampleEnvironment goal succeeds only if the entire area is sampled within the specified time, however, actual detection of contaminants is not required.

A Search goal directs the vehicle to conduct a search of the operating area using all available sensors. The "requiredPD" and "datumType" attributes specify the required probability of detection of the search and whether the search is to focus on the centroid of the area or cover the entire area equally. An optional "singleTarget" (default

of “false”) attribute specifies whether or not there is a possibility of multiple targets. The search objectives are specified using zero or more “Target” child elements. If no “Target” elements are utilized, the vehicle defaults determine the search objectives. A Search goal with a “singleTarget” attribute value of “true” succeeds immediately if the search objective is located. If the “singleTarget” attribute is “false,” the goal will succeed if the entire operating area is searched regardless of the number of objectives located.

F. MISSION RESULTS

1. Overview

Like mission definitions, mission results are encoded in an AVCL document with an “AVCL” root element. Results data is contained in “MissionLog” and “MissionResults” elements that follow the “MissionPreparation” element as depicted in Figure A.5. The “MissionLog” element contains discrete events that might occur arbitrarily over the course of the mission. The “MissionResults” element, on the other hand, contains sampled vehicle state information (i.e., telemetry and control orders) from various points in the mission. Examples include target detections, message receipts and transmissions, and behavior activations. The content of these two elements is discussed in the following sections.

2. Discrete Event Logging

The “MissionLog” element contains a sequence of zero or more “UUVEvent,” “UGVEvent,” “USVEvent” or “UAVEvent” elements (element name corresponds to the vehicle type), each containing a single child element corresponding to an event. Unlike other AVCL elements, the “timeStamp” attribute of each of these elements is required in order to facilitate post-mission data analysis. Acceptable child elements include task-level behaviors appropriate for the vehicle-type (to facilitate a behavior-activation timeline) and inter-vehicle messages (described in Section G) as a means of recording message receipt and transmission activity. Additionally, the five elements depicted in Figure A.19 are available to log other potential events of interest (particularly those relating to the accomplishment of declarative mission goals).

AVCL “Contact” elements describe targets that are detected or tracked over the course of a mission. Content includes required “name” and “contactID” attributes (xsd:string and positiveIntType respectively) identifying the contact. The “contactID”

attribute is intended to uniquely identify a particular target, so multiple "Contact" elements with the same "contactID" value refer to the same actual contact. Child elements include a required element from the AbsoluteHorizontalPositionElements group and an optional element from the VerticalBlockElements group that describe the contact's location at time it is logged. Additional optional elements include "Heading" (headingElementType) and "Velocity" (nonNegativeScalarElementType) elements that describe the vehicle's motion when the event is logged.

A "Contaminant" element records information about a chemical, radiological, biological or other contamination detected by the vehicle. A required "contaminant" attribute (contaminantType) identifies the type of contamination that was detected. A required element from the AbsoluteHorizontalPositionElements group and an optional element from the VerticalBlockElements group describe the location of the detection.

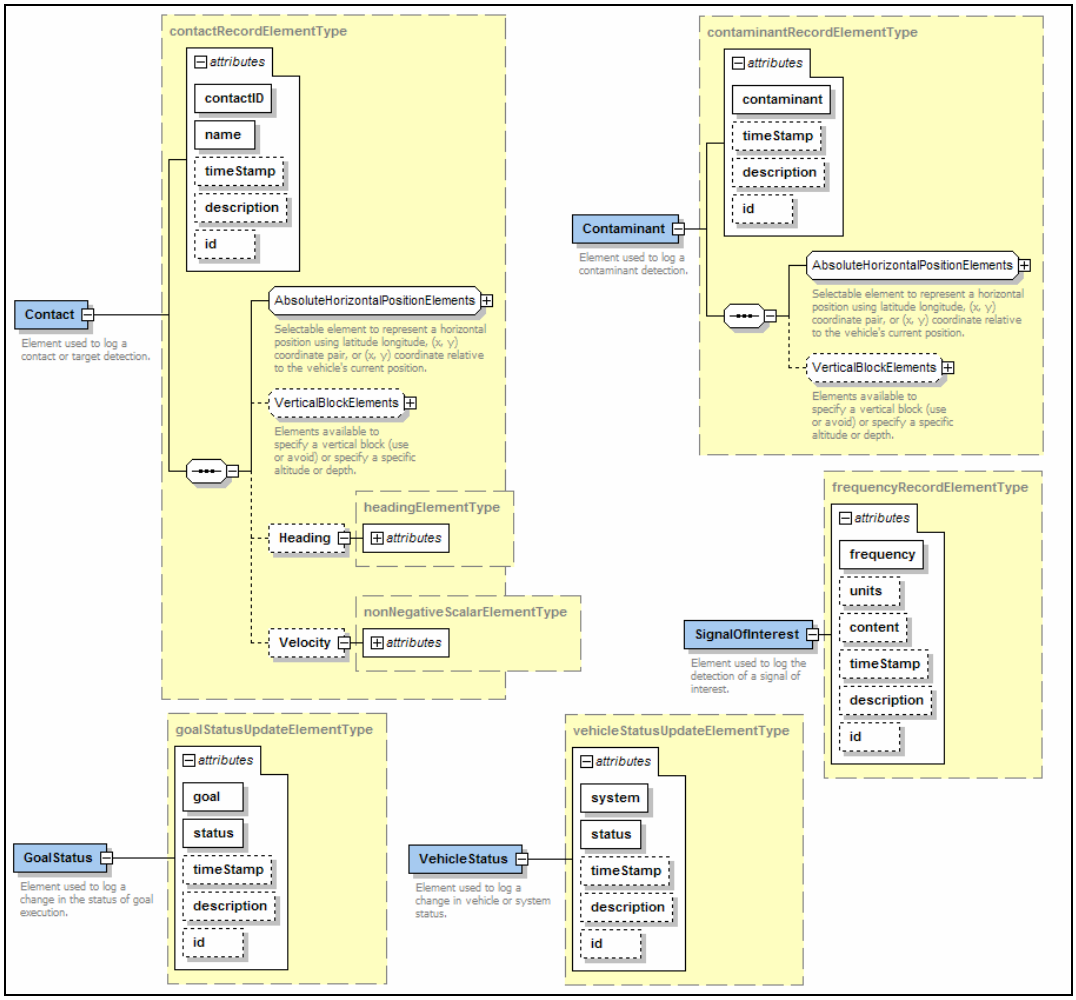


Figure A.19. AVCL Elements Used for Logging Asynchronous Discrete Events

A “SignalOfInterest” element documents the detection of electronic or acoustic transmissions. Required “frequency” (positiveScalarType) and optional “units” attributes (frequencyUnitType, default “Hz”) specify the transmission that was detected. An optional “content” element is available to include transmission content if required.

The “GoalStatus” element provides documentation of declarative agenda mission goal execution progress. A required “goal” attribute (xsd:IDREF) refers to the agenda goal and a required “status” attribute (xsd:string) contains a free-text description of the status change (e.g., “commence,” “succeed,” “fail” or “in operating area”).

Semantically similar to the “GoalStatus” element, the “VehicleStatus” element documents changes to the status of vehicle systems. Required “system” and “status” attributes (both xsd:string) identify the system in question and describe the nature of the status change (e.g., “shut down” or “low power due to overheating”).

3. Sampled Continuous Data

The “MissionResults element has two potential child elements. The first is an optional “MissionStartTime” element that contains “day” (calendarDaysType), “year” (positiveIntType), “month” (monthsType), “hour” (clockHoursType), “minute” (clockMinutesOrSecondsType), “second” (clockMinutesOrSecondsType), and “timeZone” (timeZoneType) attributes.

Following the “MissionStartTime” element (if present), are zero or more “SampledResults” elements containing sampled telemetry or control order information. As depicted in Figure A.20, this element contains a vehicle-type-specific telemetry element and / or a vehicle-type-specific control orders element.

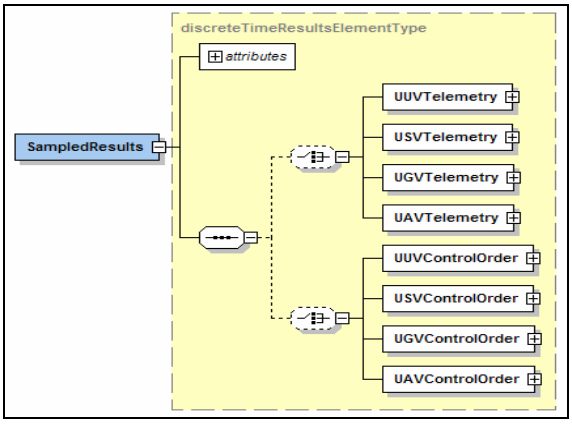


Figure A.20. An AVCL Element for Recording Vehicle Telemetry and Control Orders

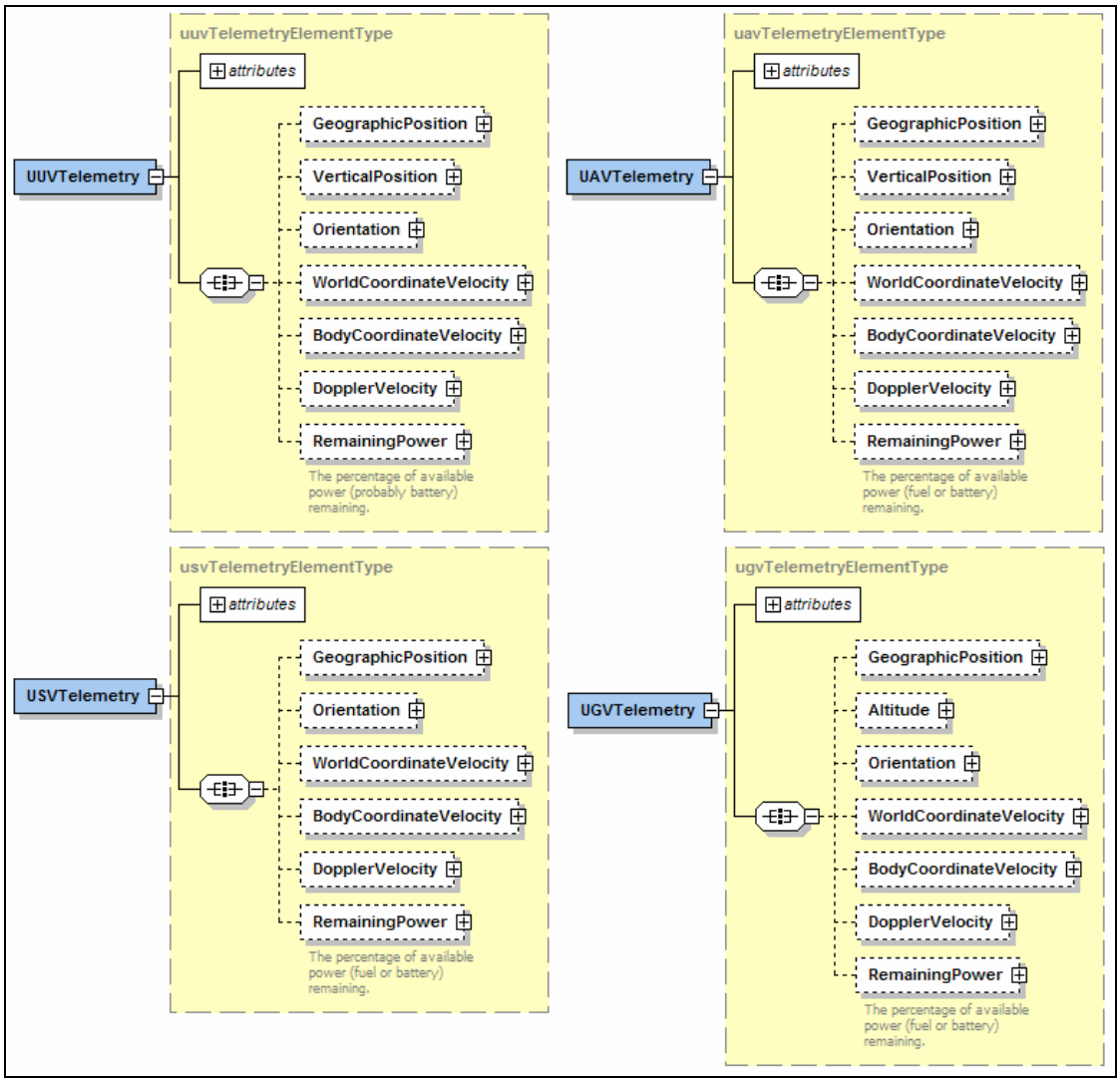


Figure A.21. AVCL Elements for Sampled Vehicle Telemetry

The content models of the vehicle-specific control-order elements are depicted in Figure A.22. As with telemetry elements, all child elements of vehicle-specific any control-order element is optional to allow for encoding arbitrary partial control-order content. The most complex content model is associated with UAVs to facilitate the element's use to encode control orders for both fixed-wing and rotary-wing UAVs. At the time of this writing, the content model for the "UGVControlOrder" element is still being developed. The attributes associated with all potential control-order element child elements are described in Table A.15.

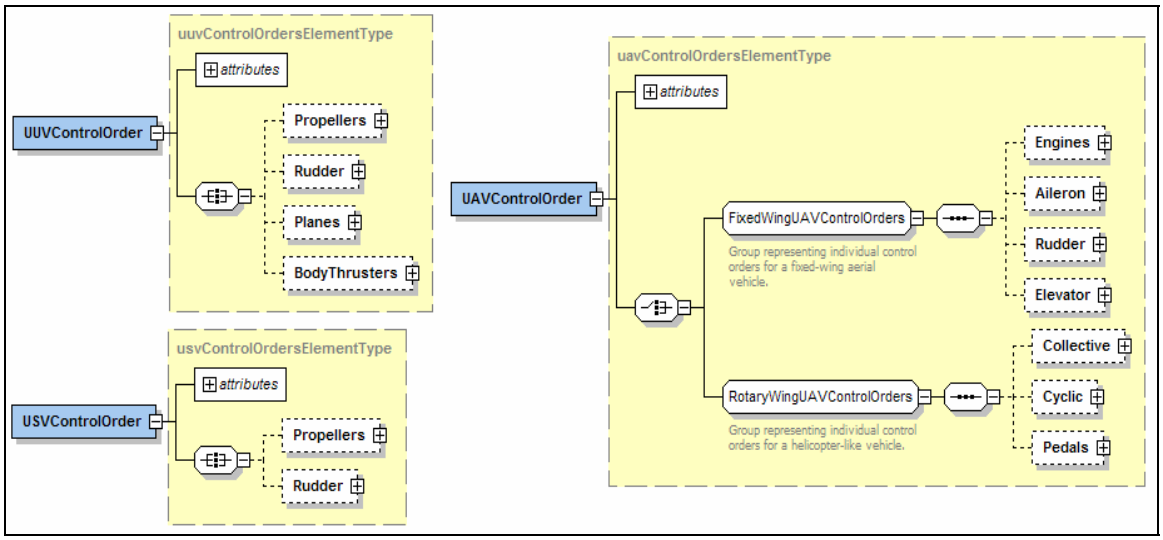


Figure A.22. AVCL Elements for Sampled Vehicle Control Orders or Settings

G. INTER-VEHICLE MESSAGING

1. Overview

Inter-vehicle messages encoded in AVCL are contained in documents with an "AVCLMessage" or "AVCLMessageList" root element. With the exception of the specific contents of the "body" child element (which contains one or more "AVCLMessage" elements), the content model of documents with an "AVCLMessageList" root element was discussed in Section D of this appendix. On the other hand, the content model of documents with an "AVCLMessage" root element was not discussed beyond the required "head" and "body" child elements. Thus, the remainder of this section focuses on the content model of these AVCL message components.

Element	Attribute	Type	Use	Description
Propellers	port	signedPercentType	Optional	Percentage of available power ordered to the port propeller.
	starboard	signedPercentType	Optional	Percentage of available power ordered to the starboard propeller.
	centerline	signedPercentType	Optional	percentage of available power ordered to the centerline propeller.
Rudder	value	signedPercentType	Required	Ordered percentage of available rudder deflection (positive leading edge right).
Planes	stern	signedPercentType	Optional	Ordered percentage of available bow-plane deflection (positive leading edge up).
	bow	signedPercentType	Optional	Ordered percentage of available stern-plane deflection.
BodyThrusters	bowLateral	signedPercentType	Optional	Percentage of available power ordered to the bow lateral cross-body thruster (positive pushes right).
	sternLateral	signedPercentType	Optional	Percentage of available power ordered to the stern lateral cross-body thruster.
	bowVertical	signedPercentType	Optional	Percentage of available power ordered to the bow vertical cross-body thruster (positive pushes down).
	sternVertical	signedPercentType	Optional	Percentage of available power ordered to the stern vertical cross-body thruster.
Engines	engine1	percentType	Required	Percentage of available power ordered to the number 1 (or only) engine of a UAV.
	engine2	percentType	Optional	Percentage of available power ordered to the number 2 engine of a UAV.
Aileron	value	signedPercentType	Required	Ordered percentage of available aileron deflection (positive induces right roll).
Elevator	port	signedPercentType	Required	Ordered percentage of available port elevator or stabilator deflection (positive leading edge up).
	starboard	signedPercentType	Required	Ordered percentage of available starboard elevator or stabilator deflection.
Collective	value	percentType	Required	Ordered percentage of available collective (power) for a rotary wing UAV.
Cyclic	longitudinal	signedPercentType	Required	Ordered percentage of fore-aft cyclic (positive forward) for a rotary wing UAV.
	lateral	signedPercentType	Required	Ordered percentage of left-right cyclic (positive right) for a rotary wing UAV.
Pedals	value	signedPercentType	Required	Ordered percentage of available tail-rotor authority (positive right) for a rotary wing UAV.

Table A.15. Attributes Associated with AVCL Vehicle-Specific Control-Order Elements

2. The AVCL Message Header

The "head" child of an "AVCLMessage" element contains the message header and complies with the content model depicted in Figure A.23. Zero or more "meta" elements (complying with the content model of previously discussed "meta" and "MetaCommand" elements) provide arbitrary descriptive information about the message. An optional "Priority" element (priorityElementType) specifies the importance of the message. If this element is omitted, the message is assumed to be of routine priority. A "Sender" element (positiveIntegerElementType) identifies the vehicle from which the message originates. Zero or more nonNegativeIntegerElementType "Recipient" elements identify the vehicles to which the message is being transmitted. A "value" attribute of "0" in a "Recipient" element identifies a broadcast message that provides information of potentially global interest. Finally, an optional acknowledgeElementType "Acknowledge" element specifies the receiving vehicles' acknowledgement requirements. Omission of the "Acknowledge" element indicates that the message does not require acknowledgement.

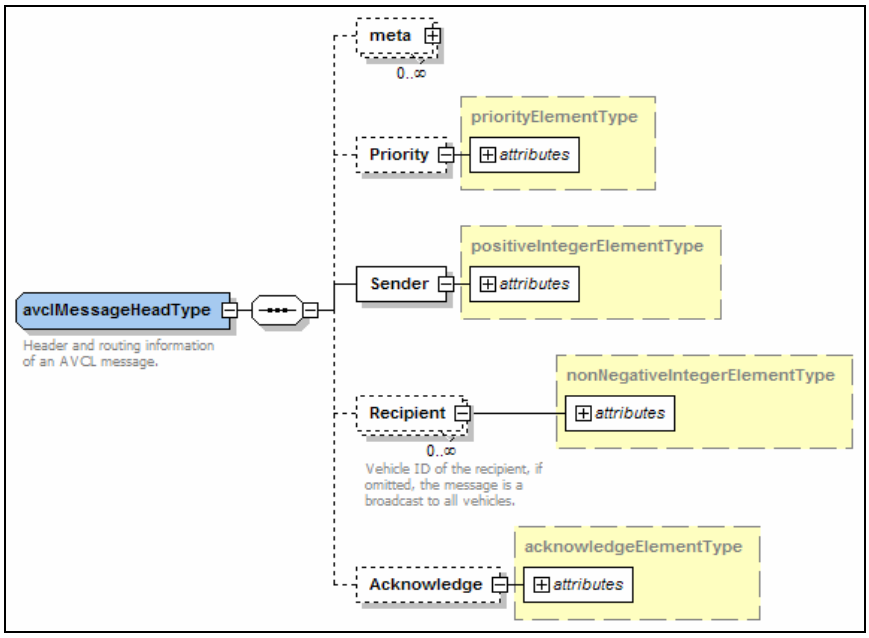


Figure A.23. Content Model of the AVCL Message Header

3. The AVCL Message Body

The content model of the AVCL message "body" element consists of a single child element from the choices depicted in Figure A.24. This element contains the

information that is to be conveyed by the inter-vehicle message. The specific content models of the individual child elements have been discussed previously in various portions of this appendix, so the intent of the remainder of this section focuses on the semantics of the various potential child elements.

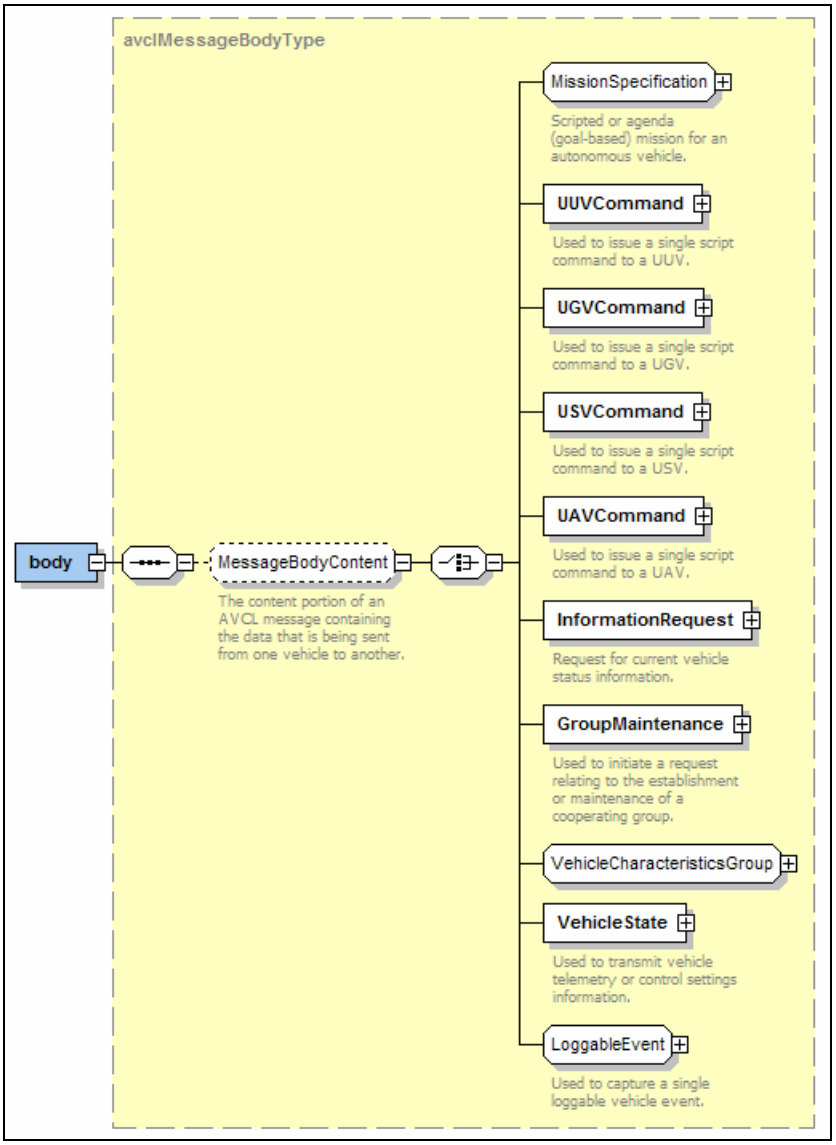


Figure A.24. The AVCL Message "body" Element Content Model

An element from the MissionSpecification group ("UUVCommandScript," "UGVCommandScript," "USVCommandScript," "UAVCommandScript," or "AgendaMission") directs the recipient vehicle to execute the contained task-level behavior script of agenda mission. Similarly, a vehicle-type-specific command element

(e.g., “UUVCommand”) containing a single task-level behavior requests that the recipient activate the contained behavior. In both cases, the receiving vehicle is required to comply with the request only if able to do so and only if the sending vehicle has the authority to direct the activity.

As the name implies, a message body containing an “InformationRequest” element requests information from the receiving vehicle rather than activity. The type of information requested is identified with a required informationRequestType “value” element. Unlike the mission-execution or behavior-request messages, a vehicle receiving an information request message is to provide the requested information if able regardless of the authority of the sending vehicle.

The final request-type message per (FIPA, 02) in the AVCL vocabulary uses a “GroupMaintenance” element with a required “request” attribute (groupMaintenanceType). The response to a group maintenance message is currently under development, but is expected to be along the lines of the protocols described and proposed by CoDA project in (Chappell, et al., 97) and (Turner and Turner, 04).

Inform messages include those conveying vehicle characteristics, vehicle state and events of interest. Vehicle characteristics are provided in messages with a “UUVCharacteristics,” “UGVCharacteristics,” “USVCharacteristics,” or “UAVCharacteristics” element with the same content model as the vehicle-characteristics child element used in the “MissionPreparation” element (Figure A.6). A “VehicleState” element complies with the content model of the “SampledResults” element depicted in Figure A.20 and provides information about the sending vehicle’s current telemetry or control status. Finally, the content models of the “UUVEvent,” “UGVEvent,” “UGVEvent” or “UAVEvent” elements are identical to the same-named child elements of the “EventLog” element discussed in Section A. and convey information about asynchronous events that occur over the course of a mission.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: THE AUTONOMOUS AND UNMANNED VEHICLE WORKBENCH (AUVW)

A. INTRODUCTION

The AUVW is an ongoing NPS project aimed at addressing the paucity of vehicle-independent planning systems. Designed around the functionality provided by AVCL, the AUVW is a Java application for UAV, USV, UGV and UUV mission-planning, mission-rehearsal and mission-playback. Features include a geographically synchronized two-dimensional graphical user interface for mission development and editing, physics-in-the-loop mission rehearsal using six-degree-of-freedom models, two-dimensional and three-dimensional visualization of mission progress, import and export of vehicle-specific data, and networked communication between the AUVW and vehicles before, during and after mission execution.

AUVW features utilize the Java Look + Feel guidelines to achieve cross-platform compatibility. Additionally, the JavaHelp system has been utilized to provide extensive documentation in a context-sensitive manner throughout the workbench toolbars and buttons. Details of the initial AUVW implementation and capabilities are documented extensively in (Lee, 04). Additional functionality is discussed in (Davis and Brutzman, 05). This appendix summarizes relevant portions of these documents and provides an overview of updated AUVW features as well. Topics include an overview of AUVW mission-planning functionality, mission-rehearsal capability, the use of X3D graphics and the DIS protocol to support visualization, and components supporting operation and interaction with actual vehicles and operators.

B. SCRIPTED MISSION PLANNING AND EDITING

The AUVW supports development of scripted autonomous vehicle missions using AVCL task-level behaviors. Multiple graphical user interface displays are provided that allow for adding new task-level behaviors to a script and editing existing behaviors. Additionally, the editor supports simultaneous testing and editing of multiple missions. An example of this functionality is depicted in Figure B.1.

The AUVW provides three behavior-level display formats for viewing AVCL task-level script missions, two of which are depicted in Figure B.2. The view providing

the most editing functionality is the icon view which uses a list of icon / name pairs to graphically depict the task-level script. Individual behaviors can be added to the end of the mission or inserted anywhere in the mission using either a pulldown or a popup menu. Behavior-type-specific dialog boxes similar to the one depicted in Figure B.3 are used to edit individual commands. These dialog boxes are activated via the same menus or by double-clicking the command in the icon list. Individual commands can be deleted, copied, or moved to new locations in the script using pulldown or popup menus or user-specified hot keys. Additionally, the popup and pulldown menus provide the capability to add general mission metadata or set the mission's geographic origin (i.e., the geographic position of the origin of the earth-fixed Cartesian coordinate system utilized throughout the mission file).

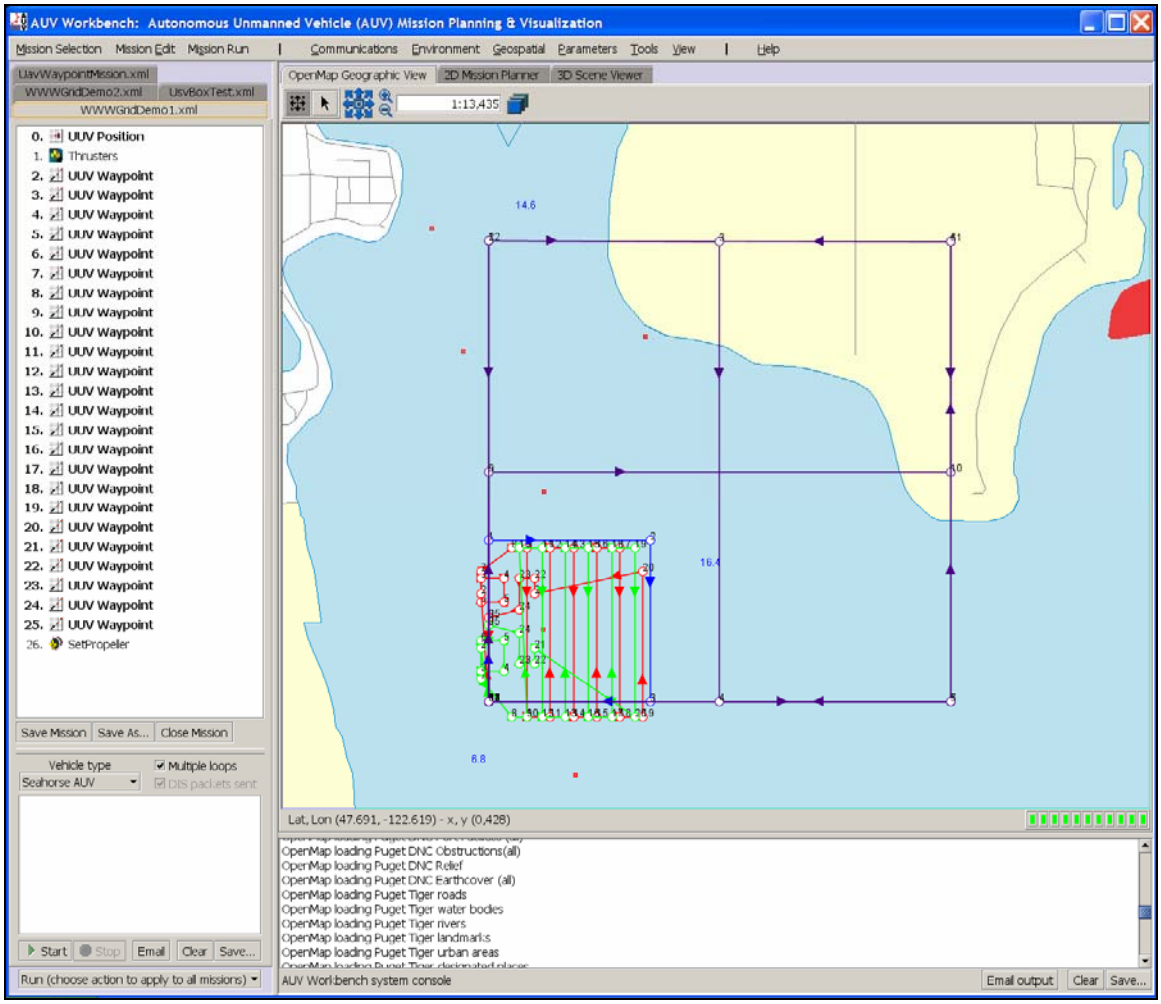


Figure B.1. Screen Snapshot of the AUVW being used to Simultaneously Edit Scripted UAV, USV and UUV Missions (From: Davis and Brutzman, 05)

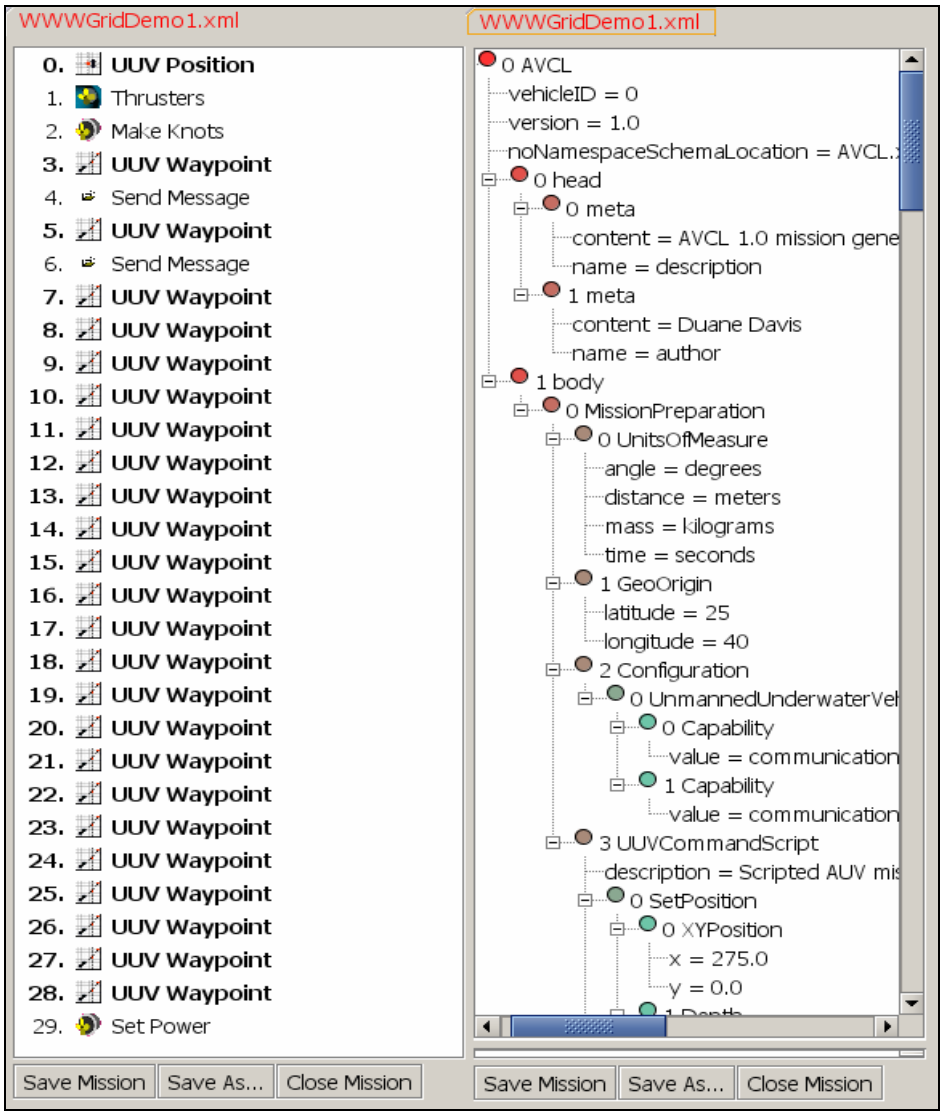


Figure B.2. AUVW Icon and Tree Views of an AVCL Task-Level Behavior Script

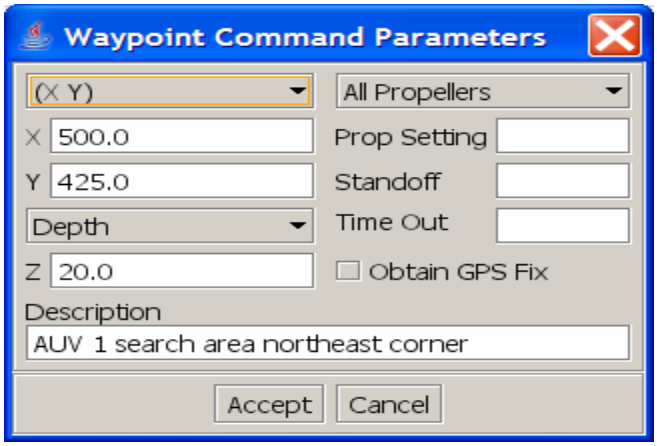


Figure B.3. AUVW Dialog Box for Editing UUV Waypoint Behaviors (From: Davis and Brutzman, 05)

Although capable of accessing all AUVW task-level mission editing functions, the icon view does not depict the full AVCL document. Header information, metadata, and mission-results elements, for instance, are not displayed. The tree and text views, however, can be utilized to view portions of the AVCL document not accessible from the icon view. An example of an AVCL task-level behavior script in the tree view is depicted in Figure B.2. In the present AUVW implementation, the tree and text views cannot be used for script editing.

While the icon, tree and text views provide the capability to view and edit all portions of an AVCL task-level behavior script, a geographic interface is a potentially more intuitive means of generating and editing autonomous vehicle missions. The AUVW, therefore, provides multiple two-dimensional interfaces that complement the functionality of the other display methods. The first displays the mission tracks of currently loaded missions on a Cartesian grid with the positive-X axis oriented true north and the positive-Y axis oriented true east. Locations can be entered using Cartesian coordinates that are plotted directly on the two-dimensional display or latitude and longitude which are converted to Cartesian coordinates based on the user-defined geographic origin.

Figure B.4, provides the two-dimensional mission view corresponding to the icon and tree views of Figure B.2. A number of behaviors (i.e., Thrusters, MakeKnots, SendMessage and SetPower) that are depicted in the icon and tree views are not depicted in the two-dimensional view. Unlike the icon, tree, and text views, the two-dimensional planner does not display all AVCL task-level commands in the mission, but only those task-level behaviors possessing a geographic component (CompositeWaypoint, Hover, Loiter, SetPosition, and Waypoint), so full mission editing capability is not provided from the two-dimensional view. Nevertheless, these are among the most common behaviors utilized in autonomous vehicle missions, so most mission editing requirements can be accessed using the two-dimensional editor. Drag and drop, snap to grid, click to highlight, and double-click to edit features support precise graphical-user-interface-based modification of existing behaviors. In particular, the pulldown menu functionality to insert new commands (including those not visible in the two-dimensional display), copy,

move, edit or delete existing commands, add metadata or set the mission's geographic origin provide a robust task-level mission editing capability.

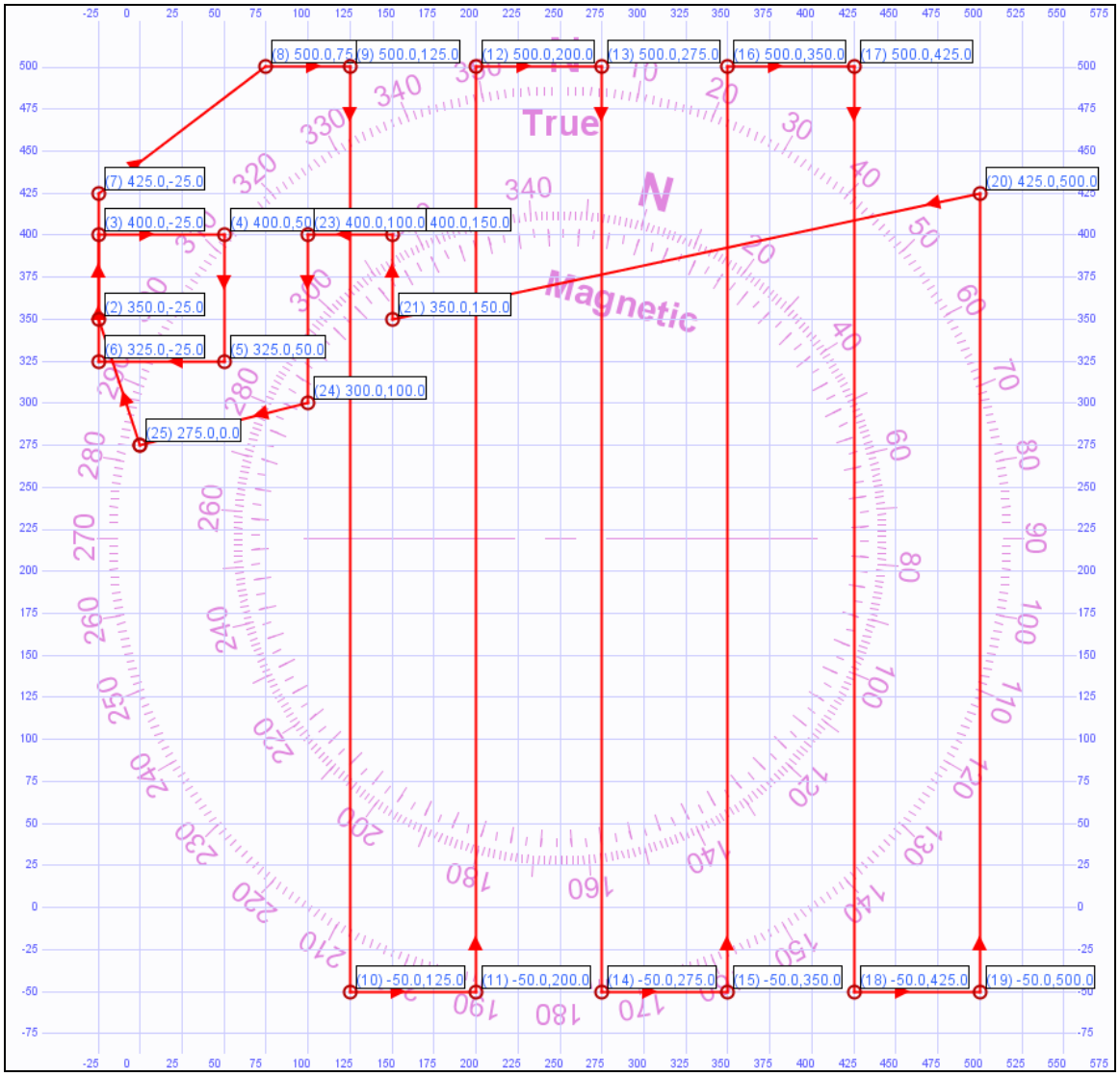


Figure B.4. The AVCL Task-Level Behavior Script Corresponding to Figure B.2 Depicted in the AUVW Two-Dimensional Cartesian Coordinate-Based Editing Interface (From: Davis and Brutzman, 05)

The final mission-editing display available in the AUVW is the geographically-based OpenMap™ editor, an exemplar view of which is provided in Figure B.5. OpenMap™ is an open source Java API for handling geospatial data and digital map data (BBN, 01). The AUVW utilizes U.S. Census Bureau Census 2000 Tiger/Line data in shapefile format (ESRI, 98) and Digital Nautical Charts to enable users to plan missions for a specific geographic area. OpenMap™ allows the user to selectively enable and

disable dataset layers, so the display can be manipulated to include as much or as little geographic information as desired.

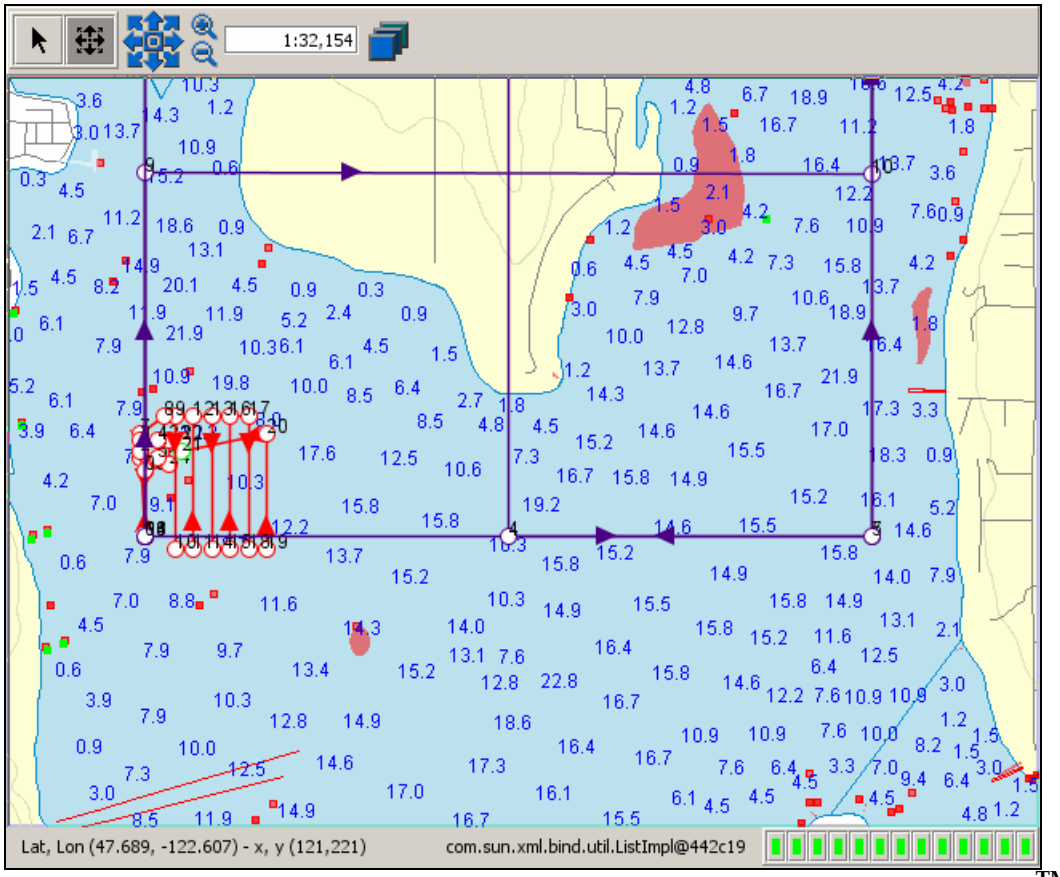


Figure B.5. UUV and UAV Task-Level Behavior Scripts Depicted in the OpenMap™ Editing Interface (From: Davis and Brutzman, 05)

As with the two-dimensional editor, only those task-level behaviors having a geographic component are overlaid in the OpenMap™ editor. Still under development, the OpenMap™ editor provides only limited mission editing capability when compared with other AUVW modes. At present, physical manipulation of task-level missions using the OpenMap™ editor is limited to drag and drop repositioning of individual behavior locations. Additionally, all previously discussed pulldown menu functionality can be accessed while using the OpenMap™ editor. Ultimately, it is envisioned that this editor will closely mirror, and possibly replace, the current AUVW 2two-dimensional planner.

C. DECLARATIVE MISSION PLANNING AND EDITING

Declarative mission editing is a fairly new addition to the AUVW and relies on the same or similar editing interfaces as scripted missions. The two-dimensional editor is

available for viewing the layout of declarative missions as depicted in Figure B.6. Operating areas are displayed using translucent two-dimensional shapes while avoid areas are displayed using more opaque shapes. Launch and recovery positions (if specified) are depicted as points labeled "L" or "R" respectively. Declarative mission editing is not available through the two-dimensional display, although display-specific functionality (e.g., zoom) is available via popup menu. Additionally, OpenMap™ editor functionality has not yet been updated to incorporate declarative mission visualization.

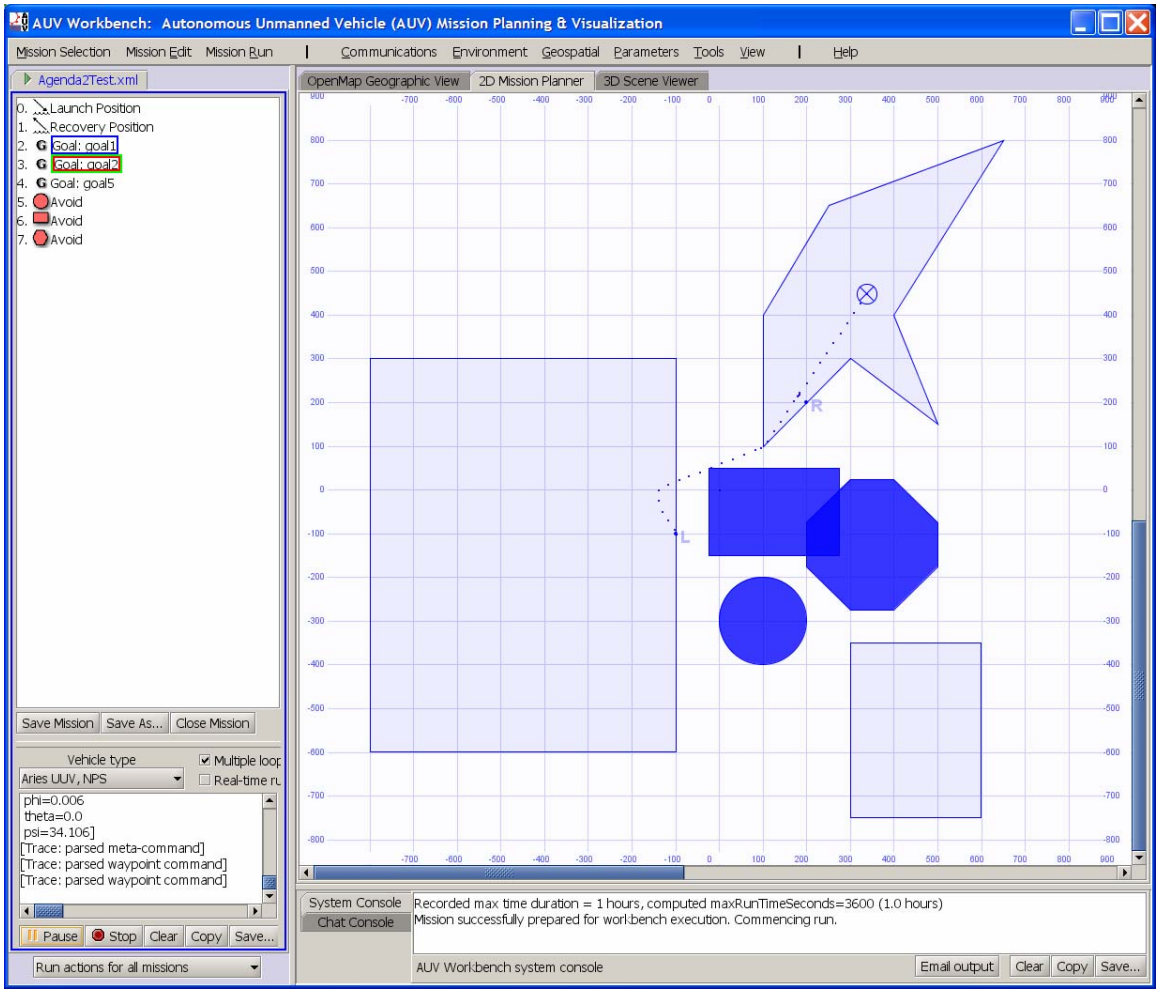


Figure B.6. An AVCL Declarative Agenda Displayed in the AUVW Two-Dimensional and Icon Views

Editing of declarative AVCL missions is conducted using an icon view specific to these types of missions (also depicted in Figure B.6). The declarative mission icon view includes entries for launch and recovery positions, goals, and avoid areas. Entries can be

added to the mission using pulldown or popup menus, and existing entries can be edited by double-clicking the entry of interest. As with the icon view associated with task-level behavior scripts, popup and pulldown menus provide the capability to add general mission metadata or set the mission's geographic origin. Additionally, the tree and text views are available when editing declarative missions as well.

Double-clicking an existing goal for editing or utilizing the pulldown or popup menu to create a new goal will activate the dialog box depicted in Figure B.7. This dialog box is used to specify all goal parameters including operating area, goal-type, goal-type-specific parameters, timing requirements, and follow-on goals upon success or failure. Similar dialog boxes are provided for creating and editing declarative mission launch and recovery positions and avoid areas.

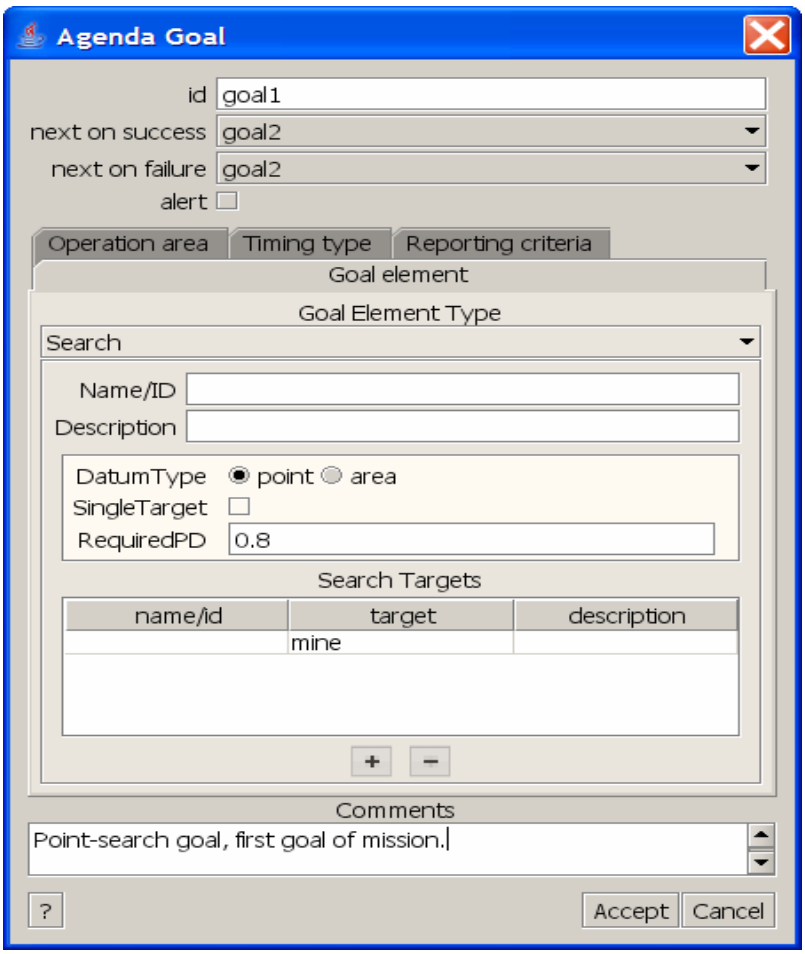


Figure B.7. An AUVW Dialog Box for Editing a Declarative Mission Goals

D. MISSION REHEARSAL

1. Simulation in the AUVW

Among the most important capabilities of the AUVW is the ability to realistically rehearse both scripted and declarative missions in a virtual environment. Mission rehearsal utilizes physically-based models that accurately represent the vehicles for which the missions are being designed. At present, UUV, USV and UAV models are implemented. Model descriptions can be found in (Brutzman, 94), (Cooke, et al., 92) and in Chapter VIII of this dissertation. All simultaneous simulated missions run in the same virtual environment enabling the operator to determine the effectiveness of multi-vehicle plans.

Simulations can be constrained to run in real time or accelerated to improve performance. During faster-than-real-time simulations involving multiple vehicles, synchronization is maintained by consistently matching all vehicle speedup factors (e.g., 50 times real time). In this way synchronization can be maintained not only among vehicle simulations running within a single AUVW session, but among those spawned by other, possibly distributed, AUVW instances as well.

2. Environmental Modeling

Proper modeling of environmental factors can produce major changes in sensor propagation, vehicle buoyancy, vehicle control and predicted power consumption. Therefore, multiple environmental datasets and services are being connected to the AUVW in order to maximize the real-world physics modeling capability for mission rehearsal and mission evaluation.

The AUVW includes the ability to read supercomputer-generated Network Common Data Form (Rew, et al., 05) datasets which include four-dimensional (x y z t) gridded time series of ocean parameters such as sound speed profile, local ocean current and wind speed. Similar real-time oceanographic parameters are also available via XML-based mechanisms including web-service queries to Fleet Numerical Meteorological Oceanographic Center computers assets.

3. Visualization

The AUVW supports three-dimensional visualization of mission progress during mission rehearsal and playback through the use of X3D—an International Organization

for Standards (ISO) standardized format for web-capable three-dimensional graphics. It utilizes an XML-enabled file format to facilitate the transfer of three-dimensional graphics data across networked applications. Significantly more robust than its web-capable three-dimensional predecessor, the Virtual Reality Modeling Language, X3D includes implicit support for DIS networking and incorporates a rigorously defined Scene Access Interface making it well-suited for use in the AUVW.

X3D-based three-dimensional visualization is implemented in the AUVW with the Xj3D toolkit (Hudson, 04). Xj3D is an open source API produced by Yumetech, Inc. for developing applications utilizing X3D content. Implemented with the support of the Web3D Consortium as an exemplar X3D-compliant browser, Xj3D implements most aspects of the interchange, interactive and immersive X3D profiles (ISO and IEC, 04) as well as a number of proposed extensions to the ISO standard. Figure B.8 shows the AUVW Xj3D viewer being used to monitor the operations of multiple UUVs in a virtual environment incorporating bathymetry and cartography near Panama City, Florida.

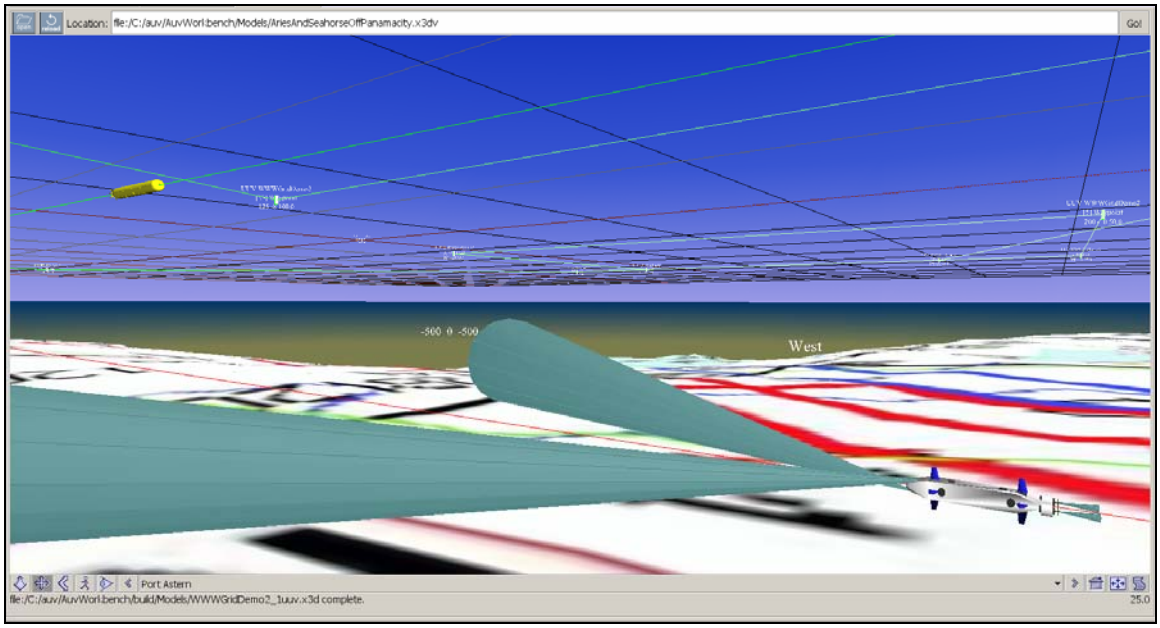


Figure B.8. ARIES and Seahorse UUVs Operating in the Same Virtual Environment as Seen in the AUVW Xj3D Viewer (From: Davis and Brutzman, 05)

A number of vehicle and virtual environment models are included in the AUVW distribution. In addition, approximately 1000 models including vehicles, structures, sensors, terrain and even entire scenarios are available for unencumbered individual

government and corporate use in the Scenario Authoring and Visualization for Advanced Graphical Environments online archive (available at https://savage.nps.edu/Savage/). A number of authoring tools are also available that facilitate the use of these and other models. Thus, the development of large virtual environments remains time consuming, but is becoming a more straightforward process. When coupled with the potential autogeneration of significant virtual environment content, the rapid creation of realistic virtual environments to rehearse and visualize real-world operations of arbitrary autonomous vehicles is becoming an achievable goal.

4. The X3D Scene Access Interface

Among the most important Xj3D features is implementation of the X3D Scene Access Interface—a portion of the X3D specification that provides for programmatic access to a loaded scene graph. Within the AUVW, the Scene Access Interface enables dynamic generation of X3D content for addition to the existing virtual environment as well as the manipulation of existing content

The first implemented AUVW dynamic generation of content using the Scene Access Interface takes the form of mission-path trackline addition to the scene. Mission tracks are automatically created based on the content of the activated AVCL mission using X3D indexed line sets and billboards when the mission is loaded for rehearsal. This X3D content corresponding to a mission's path is generated by applying an XSLT stylesheet to the AVCL document as described in the next section. The resultant X3D is then added to the current virtual environment scene graph using the X3D Scene Access Interface. If the mission is subsequently edited and rerun, the previously generated content is removed from the scene graph and replaced with updated content.

A second use of the Scene Access Interface for manipulation of the AUVW virtual environment is its use for sensor modeling. The virtual environment contains all of the objects with which the autonomous vehicles are intended to interact during mission rehearsal and playback. Enough information is therefore contained in the scene graph for vehicles operating in the virtual environment to model various sensors through the use of collision detection and picking. (Davis, 96) documented the use of C++ and the Open Inventor™ SoRayPickAction (OIAG, 94) to model mechanically steered narrow-beam active sonars installed on the NPS Phoenix UUV. Unfortunately, the X3D specification

does not support general collision detection along required for ray-picking operations. Xj3D, however, implements a proposed formal extension to the X3D specification that supports various forms of picking suitable for sensor modeling (Yumetech, 04). Specifically, the AUVW uses the Xj3D PrimitivePicker node to obtain the functionality provided by the Open Inventor™ SoRayPickAction. Individual nodes are created and added to the virtual environment using the Scene Access Interface upon the request of individual vehicle instances. Each vehicle can manipulate pickers via the Scene Access Interface as required to model onboard sensors. Individual sensors can be modeled with single picker nodes (as depicted in Figure B.9) or multiple nodes depending on sensor characteristics. Currently implemented vehicles use this functionality to model fathometers, sonar and radar altimeters, and ranging sonars.

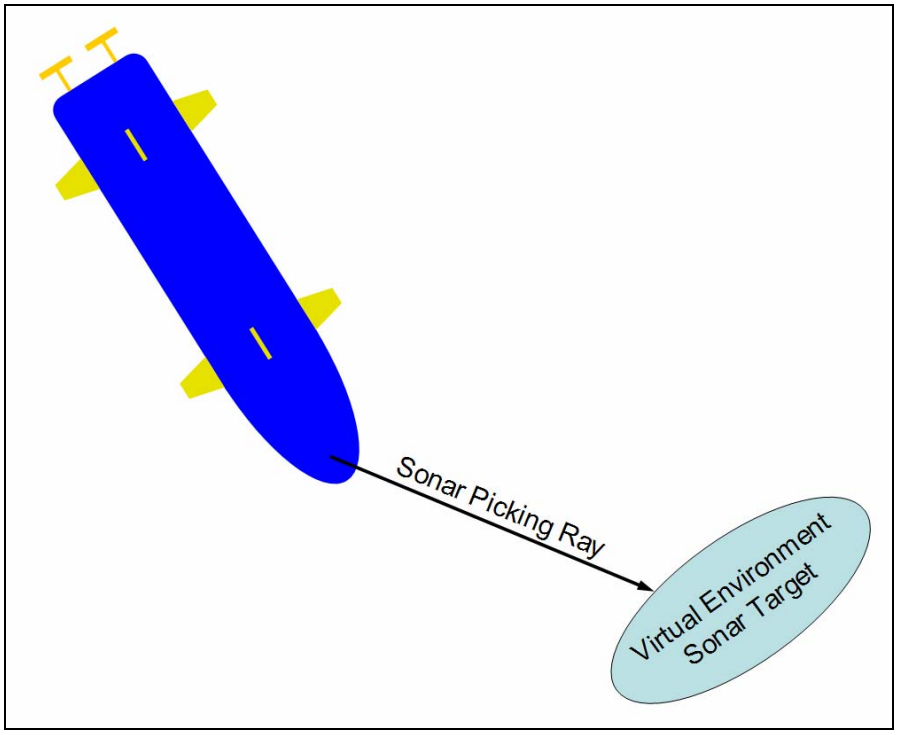


Figure B.9. Sensor Modeling using the X3D Scene Graph and Xj3D Picking Nodes (From: Davis and Brutzman, 05)

5. Distributed Interactive Simulation (DIS)

Vehicle position in the virtual environment is maintained through the use of DIS updates with individuals periodically transmitting multicast entity state protocol data units. Explicitly supported by X3D, entity state protocol data units are transmitted using

multicast and provide a means of simultaneously updating multiple views into a common virtual environment. This inherently supports the use of multiple AUVW instances in a networked environment to provide for planning and rehearsal of multi-vehicle missions from different locations by synchronizing the virtual environment across the network.A recent addition to the AUVW is support for DIS XML. This addition uses an Extensible Messaging and Presence Protocol (XMPP) channel to exchange entity state protocol data units encoded with XML (McGregor, et al., 06). An Xj3D browser extension provides direct access to the XMPP channel from within the X3D scene graph and automatic monitoring of DIS XML packets. Additional XMPP support provides AUVW user access to chat rooms through which DIS XML protocol data units pass.

DIS support highlights an additional planned use of the Scene Access Interface: DIS entity monitoring to support automatic addition of applicable vehicle models to the scene graph. When a protocol data unit is received, its content can be analyzed to determine whether or not its model is already contained in the virtual environment's scene graph. If not, the protocol data unit's siteID, applicationID, entityID and marking fields provide enough information to uniquely identify the type of vehicle. With this information, an instance of the appropriate vehicle that will correctly respond to subsequent entity state protocol data units can be created or loaded dynamically and inserted into the virtual environment scene graph.

E. VEHICLE SUPPORT

1. Data Format Conversion

The AUVW provides for the automated generation of vehicle-specific missions from task-level behavior scripts with XSLT stylesheets as described in Chapter V. The AUVW conducts XSLT transformations using the Xalan-Java API, an open source product of the Apache XML Project (Apache, 06). The graphical user interface panel of Figure B.10 (activated via pulldown menu) is used to initiate the transformation of stored or loaded task-level behavior scripts as required.

The AUVW also allows the import of vehicle-specific missions through context-free-grammar-based conversion to task-level behavior scripts as described in Chapter V. This functionality is accessed via a pulldown menu providing for selection of the name and vehicle-specific format of import file. As of this writing, mission import and export

support is provided for the NPS Phoenix and ARIES UUVs, the Naval Oceanographic Office Seahorse UUV, the Hydroid REMUS UUV and the JAUS message set.

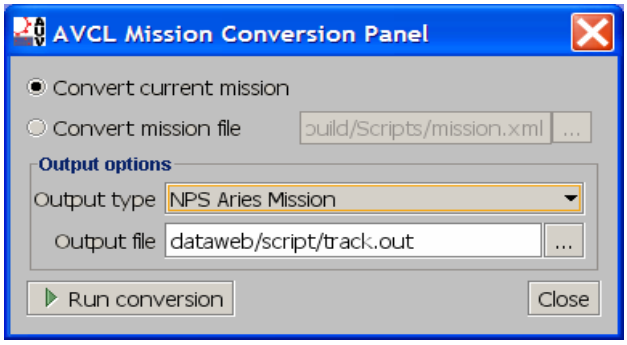


Figure B.10. Support for Automated Conversion of Task-Level Behavior Scripts to Vehicle-Specific Formats Using XSLT Stylesheets (From: Davis and Brutzman, 05)

2. Communications

All phases of autonomous vehicle operations generally require some level of communication between vehicles and operators. During the pre-mission phase, the operator must be able to initialize the vehicle and load and initiate missions. During mission execution, many vehicles are able to provide position and status reports or receive updated tasking. Following execution, mission results must be downloaded from the vehicle to offboard systems for analysis and archiving. The AUVW has a number of communications capabilities implemented or planned to support these requirements.

Communications involving autonomous vehicles routinely utilize acoustic modems or other devices relying on serial communications. The AUVW implements user-configurable serial communications and Kermit protocol file transfer appropriate for point-to-point communications between the AUVW and a variety of devices. Also slated for implementation are File Transfer Protocol, Secure File Transfer Protocol, Terminal Emulation and Secure Shell facilities that will improve the flexibility and efficiency of communications between the AUVW and controlled vehicles that use Transmission Control Protocol / Internet Protocol networking.

Collaboration support is provided through the implementation of XMPP-based chat and Hypertext Transfer Protocol (HTTP) server support. XMPP-based chat provides for real-time communication between distributed AUVW users and also provides

infrastructure for the transfer of DIS XML packets. HTTP support, on the other hand, provides for data sharing between various locations.

F. AVAILABILITY AND DEVELOPMENT

Due to the consistent use of XML, Java and Java Look + Feel, the AUVW has been successfully tested on Windows, MacOSX, Linux and Solaris systems. An online autoinstaller is updated weekly, and installation Digital Versatile Discs (DVD) are available on request.

Source code is available under an open source license that ensures unencumbered use by individuals, government projects and industry. All source code, configuration files and documentation are maintained under Concurrent Version System control, allowing around-the –clock distributed development by qualified participants. Additionally, an archived mailing list is used to discuss design issues and problem resolution. Finally, the Bugzilla tracking system is used to resolve all problems and precisely define new features. Each of these resources and additional documentation is available online at https://www.movesinstitute.org/xmsf/xmsf.html#Projects-AUV.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Abbott, I. H. and Von Doenhoff, A. E., *Theory of Wing Sections*, Dover Publications, 1959.

Ahner, D. K., "Planning and Control of Future Combat System UAVs," 72nd Military Operations Research Society Symposium, Unmanned Systems Working Group Presentation, Monterey, California, June 2004.

Albus, J. S., "Task Decomposition," *Proceedings of the 8th IEEE International Symposium on Intelligent Control*, August 1993.

Albus, J. S., "A Reference Model Architecture for Intelligent Hybrid Control Systems," *Proceedings of the 1996 Triennial World Congress, International Federation of Automatic Control*, July 1996.

Albus, J. S., "Engineering Intelligent Systems," *Proceedings of the ISIC/CIRA/ISAS 1998 Conference*, September 1998.

Apache Software Foundation, Xalan-Java 2.7.0 Online Documentation, 2006. Available at http://xml.apache.org/xalan-j/index.html. Accessed August 2006.

Arcineas, F., *C++ XML*, New Riders Publishing, 2002.

ASTM Work Item WK9139, *Standard Guide for Unmanned Undersea Vehicle Mission Payload Interface*, expected publication in 2006.

BBN Technologies Solutions, LLC, OpenMap Viewer Application User's Guide, November 2001. Available at http://openmap.bbn.com/doc/user-guide.html. Accessed August 2006.

Blidberg, D. R., "Generic Behaviors: Definition and Structure, an Approach to Modularity in Intelligent System Control Architectures—Volume 1: Technical Proposal," Proposal for SOL BAA #94-19, July 21, 1994.

Bourg, D. M., *Physics for Game Developers*, O'Reilly and Associates, 2002.

Brutzman, D. P., *A Virtual World for an Autonomous Underwater Vehicle*, Ph.D. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, December 1994. Available at http://web.nps.navy.mil/~brutzman/dissertation. Accessed August 2006.

Buzzell, C. M., *A Common Control Language for Multiple Autonomous Undersea Vehicle Cooperation*, Master's Thesis, University of Massachusetts Dartmouth, October 2004.

Byrnes, R. B., *The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for Control of Autonomous Vehicles*, Ph.D. Dissertation, Department of Computer Science, Naval Postgraduate School, Monterey, CA, March 1993. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/dissertations. Accessed August 2006.

Chappell, S. G., Turner, R. M., Turner, E. H. and Grunden, C., "Cooperative Behavior in an Autonomous Oceanographic Sampling Network: MAUV Project Update," *Proceedings of the 10th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, September 1997.

Chief of Naval Operations (CNO), *Navy Search and Rescue Tactical Information Document (SAR TACAID)*, U.S. Navy Naval Warfare Publication (NWP) 3-22.5-SAR-TAC, 1997.

Chief of Naval Operations, *Integrating Unmanned Vehicles into Maritime Operations*, U.S. Navy Tactical Memorandum TM 3-22.5-SW, 2004.

Cooke, J. M., Zyda, M. J., Pratt, D. R. and McGhee, R. B., "NPSNET: Flight Simulation Dynamic Modeling Using Quaternions," *Presence*, v. 1, number 4, pp. 404-420, Fall 1992.

Corman, T. H., Leiserson, C., E. and Rivest, R. L., *Introduction to Algorithms*, McGraw-Hill Book Company, 1990.

Crangle, C. and Suppes, P., *Language and Learning for Robots*, Center for the Study of Language and Information Publishing, 1994.

Daconta, M. C., Obrst, L. J. and Smith, K. T., *The Semantic Web, A Guide to the Future of XML, Web Services, and Knowledge Management*, Wiley Publishing, 2003.

Davis, D. T., *Precision Maneuvering of the Phoenix Autonomous Underwater Vehicle for Entering a Recovery Tube*, Master's Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, September 1996. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/theses. Accessed August 2006.

Davis, D. T., "Automated Parsing and Conversion of Vehicle-Specific Data into Autonomous Vehicle Control Language using Context-Free Grammars and XML Data Binding," *Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/papers. Accessed August 2006.

Davis, D. T. and Brutzman, D. P., "The Autonomous and Unmanned Vehicle Workbench: Mission Planning, Mission Rehearsal, and Mission Replay Tool for Physics-Based X3D Visualization," *Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/papers. Accessed August 2006.

Defense Advanced Research Projects Agency (DARPA) and Information Society Technologies Programme (IST), DAML+OIL Language Specification, March 2001. Available at http://www.daml.org/2001/03/daml+oil-index.html. Accessed August 2006.

Department of Defense (DOD) Joint Robotics Program (JRP), *Fiscal Year 2005 Joint Robotics Program Unmanned Ground Vehicle Master Plan*, 2004. Available at http://www.jointrobotics.com/activities_new/masterplan.shtml. Accessed August 2006.

Department of the Navy (DON), *The Navy Unmanned Undersea Vehicle (UUV) Master Plan*, November 2004. Available at http://orionprogram.org/PDFs/UUV_USNavy.pdf. Accessed August 2006.

Department of the Navy, *Department of the Navy XML Naming and Design Rules*, Version 2.0. January 2005. Available at http://www.doncio.navy.mil/StoreFront/Uploads/0128YRM15241.pdf. Accessed August 2006.

Dobeck, G., Cobb, T., Weilert, D., Bills, G., Smith, C., Bryan, J., Aridges, T., Fernandez, M., Diany, C., Zurawski, B., and Bello, M., "Computer-Aided Detection Computer Aided Classification (CAD/CAC)," ONR Joint Review of Technology Applicable to Mine Countermeasures and Associated Missions, Panama City, Florida, February 2004.

Duarte, C. N., Martel, G. R., Eberbach, and E., Buzzell, C., "Talk Amongst Yourselves: Getting Multiple Autonomous Vehicles to Cooperate," *Proceedings of the 2004 IEEE/OES Autonomous Underwater Vehicles Symposium*, Sebasco, Maine, June 2004.

Duarte, C. N., Martel, G. R., Buzzell, C., Komerska, R., Mapparapu, S., Chapel, S., Blidberg, D. R. and Nitzel, R., "A Common Control Language To Support Multiple Cooperating AUVs," *Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.

Duckett, J., Griffin, O., Mohr, S., Norton, F., Stokes-Rees, I., Williams, K., Cagle, K., Ozu, N., and Tennison, J., *Professional XML Schemas*, Wrox Press, Ltd., 2001.

Eberbach, E., "$-Calculus Bounded Rationality = Process Algebra + Anytime Algorithms," *Applicable Mathematics: Its Perspectives and Challenges*, edited by Misra, J., Narosa Publishing House, 2001.

Eberbach, E., "$-Calculus of Bounded Rational Agents: Flexible Optimization as Search under Bounded Resources in Interactive Systems," *Fundamenta Informaticae*, volume 68, IOS Press, 2005.

Environmental Systems Research Institute (ESRI) White Paper, "ESRI Shapefile Technical Description," July 1998. Available at http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf. Accessed August 2006.

Fensel, D., *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer-Verlag Publishing, 2001.

Ferguson, D. and Stentz, A., "The Delayed D* Algorithm for Efficient Path Replanning," *Proceedings of the IEEE International Conference on Robotics and Automation*, Barcelona, Spain, 2005. Available at http://gs2045.sp.cs.cmu.edu/downloads/Delayed-DStar.pdf. Accessed August 2006.

Foundation for Intelligent Physical Agents (FIPA), *FIPA Communicative Act Library Specification*, December 2002. Available at http://www.fipa.org/specs/fipa00037/SC00037J.html. Accessed August 2006.

Gottgtroy, P, Kasabov, N. and Macdonell, S., "An Ontology Engineering Approach for Knowledge Discovery from Data in Evolving Domains," Knowledge Engineering and Discovery Institute, Auckland University of Technology, New Zealand, 2003. Available at http://www.aut.ac.nz/resources/research/research_institutes/kedri/downloads/pdf/datamining2003.pdf. Accessed August 2006.

Guarino, N. and Giaretta, P., "Ontologies and Knowledge Bases: Towards a Terminological Clarification," in *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*, edited by Mars, N., Amsterdam: IOS Press, 1995.

Hall, W. and Farrell, J., "Activity-Based Mission Planning and Plan Management for Autonomous Vehicles," *Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology*, Cambridge, MA, July 1994.

Halpin, T. A., "Object Role Modeling: an Overview," Microsoft Corporation White Paper, 2001. Available at http://www.orm.net/pdf/ORMwhitePaper.pdf. Accessed August 2006.

Harold, E. R. and Means, S. W., *XML in a Nutshell*, Second Edition, O'Reilly & Associates, 2002.

Hawkins, D. L. and Van Leuvan, B. C., *An XML-Based Mission Command Language for Autonomous Underwater Vehicles*, Master's Thesis, Department of Information Sciences, Naval Postgraduate School, Monterey, California, June 2003. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/theses. Accessed August 2006.

Healey, A. J., Marco, D. B., and McGhee, R. B., "Autonomous Underwater Vehicle Control coordination Using a Tri-Level Hybrid Software Architecture," *Proceedings of the IEEE Robotics and Automation Conference*, Minneapolis, MI, 1996. Available at http://web.nps.navy.mil/~me/healey/papers/fin.pdf. Accessed August 2006.

Holman, K. G., *Definitive XSLT and XPath*, Prentice-Hall PTR, 2002.

Hopcroft, J., Motwani, R., and Ullman, J., *Introduction to Automata Theory, Languages, and Computation*, Second Edition, Addison-Wesley, 2001.

Hudak, P., Courtney, A., Nillson, H., and Peterson, J., "Arrows, Robots, and Functional Programming," *Summer School on Advanced Functional Programming 2002*, Oxford University, 2003.

Hudson, A. D., "An Introduction to the Xj3D Toolkit," *Proceedings of the 9th International Conference on 3D Web Technology*, Monterey, CA, April 2004.

Hunter, D., Cagle, K., Gibbons, D., Ozu, N., Pinnock, J., and Spencer, P., *Beginning XML*, Third Edition, Wrox Press Ltd., 2004.

Hydroid, Inc., *Technical Manual Operations and Maintenance Instructions, REMUS Remote Environmental Measuring Units*, 2001.

Institute of Electrical and Electronics Engineers (IEEE) Standard 1278.1-1995, *Standard for Distributed Interactive Simulation (DIS) Application Protocols*, 1995.

International Maritime Organization (IMO) and International Civil Aviation Organization (ICAO), *International Aeronautical and Maritime Search and Rescue Manual*, London, England, 1998.

International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) International Specification 19775:200x, *Extensible 3D (X3D) International Specification*, 2004. Available at http://www.web3d.org/x3d/specifications/#x3d. Accessed August 2006.

International Telecommunication Union (ITU), *Information Technology—Generic Applications of ASN.1—Fast Infoset*, Recommended Standard X.891, May 2005.

Jarrar, M., Demey, J. and Meersman, R., "On Using Conceptual Data Modeling for Ontology Engineering," *Journal on Data Semantics (Special Issue on Best Papers from the ER, ODBASE, and COOPIS 2002 Conferences)*, volume 2800, October 2003. Available at http://www.jarrar.info/publications/default.htm. Accessed August 2006.

Joint Architecture for Unmanned Systems (JAUS) Working Group, *JAUS Domain Model*, Volume 1, Version 3.1, 9 April 2004. Available at http://www.jauswg.org/baseline/insarchive.html. Accessed August 2006.

Joint Architecture for Unmanned Systems Working Group, *JAUS Reference Architecture Specification*, Volume II, Part 1, Architecture Framework, Version 3.2, 13 August 2004. Available at http://www.jauswg.org/baseline/insarchive.html. Accessed August 2006.

Joint Architecture for Unmanned Systems Working Group, *JAUS Reference Architecture Specification*, Volume II, Part 2, Message Definition, Version 3.2, 13 August 2004. Available at http://www.jauswg.org/baseline/insarchive.html. Accessed August 2006.

Joint Architecture for Unmanned Systems Working Group, *JAUS Reference Architecture Specification*, Volume II, Part 3, Message Set, Version 3.2, 13 August 2004. Available at http://www.jauswg.org/baseline/insarchive.html. Accessed August 2006.

Joint Architecture for Unmanned Systems Working Group, *Unmanned Systems Common Service Specification*, Draft Version 1.02, 2006.

Kalinichenko, L., Missikoff, M., Schiapelli, F. and Skvortsov, N., "Ontological Modeling," *Proceedings of the 5th Russian Conference on Digital Libraries*, St. Petersburg, Russia, 2003. Available at http://synthesis.ipi.ac.ru/synthesis/publications/ontomodeling/ontomodeling.pdf. Accessed August 2006.

Kay, M., *XSLT Programmer's Reference*, Second Edition, Wiley Publishing, 2003.

Komerska, R., Blidberg, D. R., Chappell, S. G., and Peng, L., "Progress in the Development and Evaluation of a Standard AUV Command and Monitoring Language," *Proceedings of the 11th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 1999.

Komerska, R., Chappell, S. G., Peng, L. and Blidberg, D. R., "Generic Behaviors as an Interface for a Standard AUV Command & Monitoring Language, Working Draft Version A3," Autonomous Undersea Systems Institute, 3 September 1999.

Komerska, R. J., "A Proposed Standard Language for AUV Monitoring & Control, Version 2.8," Autonomous Undersea Systems Institute Technical Report TR-0509-01, September 2005.

Kwak, S. H., McGhee, R. B., and Bihari, T. E., "Rational Behavior Model: A Tri-Level Multiple Paradigm Architecture for Robot Vehicle Control Software," Naval Postgraduate School Technical Report NPSCS-92-003, March 1992.

Lee, C. S., *NPS AUV Workbench: Collaborative Environment for Autonomous Underwater Vehicles (AUV) Mission Planning and 3D Visualization*, Master's Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, March 2004. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/theses. Accessed August 2006.

Lewis, A. S. and Weiss, L. G., "Intelligent Autonomy and Performance for Coordinated Unmanned Vehicles," *Proceedings of the National Institute of Standards Technology Performance Metrics for Intelligent Systems Workshop*, Gaithersburg, MD, August 2004. Available at http://www.isd.mel.nist.gov/research_areas/research_engineering/Performance_Metrics/PerMIS_2004/Proceedings/Lewis.pdf. Accessed August 2006.

Luger, G. F., *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Fourth Edition, Addison-Wesley, 2002.

Manley, J., "Multiple AUV Missions in the National Oceanic and Atmospheric Administration," *Proceedings of the 2004 IEEE/OES Autonomous Underwater Vehicles Symposium*, Sebasco, Maine, June 2004.

Marco, D., "Procedure to Run Missions with the ARIES," Naval Postgraduate School Center for Autonomous Underwater Vehicle Research internal document, September 2001.

Marr, W. J., *Acoustic Based Tactical Control of Underwater Vehicles*, Ph.D. Thesis, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, June 2003. Available at http://www.cs.nps.navy.mil/research/auv/theses/Marr/BillMarr%20Dissertation.pdf. Accessed August 2006.

Means, W. S. and Bodie, M. A., *The Book of SAX, The Simple API for XML*, No Starch Press, 2002.

McGhee, R. B., Bachmann E. R. and Zyda, M. J., "Rigid Body Dynamics, Inertial Reference Frames, and Graphics Coordinate Systems: A Resolution of Conflicting Conventions and Terminology," Naval Postgraduate School Technical Report NPS-MV-01-002, November 2000.

McGregor, D., Brutzman, D., Armold, A., and Blais, C., "DIS-XML: Moving DIS to Open Data Exchange Standards," *Proceedings of the Simulation Interoperability Standards Organization (SISO) Spring 2006 Simulation Interoperability Workshop*, Huntsville, AL, April 2006.

McLaughlin, B., *Java & XML*, O'Reilly and Associates, 2001.

McLaughlin, B., *Java & XML Data Binding*, O'Reilly and Associates, 2002.

McLoud, T. W. and Wu, C., "Multi-Vehicle Mission Control System for Teams of Heterogeneous Unmanned Vehicles," 72nd Military Operations Research Society Symposium, Unmanned Systems Working Group Presentation, Monterey, California, June 2004.

Mendenhall, W., Wackerly, D. and Scheaffer, R., *Mathematical Statistics with Applications*, Fifth Edition, PWS-Kent Publishing, 2001.

Mitchell, T. M., *Machine Learning*, WCB/McGraw Hill, 1997.

Montgomery, D. C. and Runger, G. C., *Applied Statistics and Probability for Engineers*, Third Edition, John Wiley and Sons, 2003.

Multilateral Interoperability Programme (MIP), *The Joint C3 Information Exchange Data Model (JC3IEDM Main)*, Edition 3.0, Greding Germany, 9 December 2006.

Multilateral Interoperability Programme, *Overview of the Joint C3 Information Exchange Data Model (JC3IEDM) (JC3IEDM Overview)*, Edition 3.0, Greding Germany, 9 December 2005.

Mupparapu, S. S., Chappell, S. G., Komerska, R. J., Blidberg, R. D., Nitzel, R., Benton, C., Popa, D. O., and Sanderson, A. C., "Autonomous Systems Monitoring and Control—An AUV Fleet Controller," *Proceedings of the 2004 IEEE/OES Autonomous Underwater Vehicles Symposium*, June 2004.

National Academy of Sciences (NAS) Naval Studies Board, *Autonomous Vehicles in Support of Naval Operations*, The National Academies Press, 2005.

Naval Oceanographic Office (NAVO), "Notes on Seahorse Mission Orders File Format and Syntax," Naval Oceanographic Office internal document, 2004.

Neushul, J. D., *Interoperability, Data Control and Battlespace Visualization Using XML, XSLT, and X3D*, Master's Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, September 2003.

Nicholson, J., *Autonomous Optimal Rendezvous of Underwater Vehicles*, PhD Thesis, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, September 2004. Available at Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/dissertations. Accessed August 2006.

Object Management Group (OMG), Unified Modeling Language (UML) 2.0 Superstructure Specification, August 2005. Available at http://www.omg.org/cgi-bin/doc?formal/05-07-04.pdf. Accessed August 2006.

Open Geospatial Consortium, Inc. (OGC), OpenGIS Sensor Model Language (SensorML) Implementation Specification, edited by Botts, M., February 2006. Available at http://vast.nsstc.uah.edu/SensorML/docs/OGC-05-086r2_SensorML.doc. Accessed August 2006.

Open Inventor Architecture Group (OIAG), *Open Inventor C++ Reference Manual*, Addison Wesley Publishing, 1994.

QNX Software Systems (QNX), QNX Momentics Development Suite, 2005. Available at http://www.qnx.com/products/rtos/index.html. Accessed August 2006.

Peterson, C. A. and Head, M. E. M., "Seahorses and Submarines," *Undersea Warfare Magazine*, Volume 5, Number 1, Fall 2002. Available at http://www.navy.mil/palib/cno/n87/usw/issue_16/seahorses_and_submarines.html. Accessed August 2006.

Phoha, S. and Schmiedekamp, M., "Robotalk: A Common Control Language for Distributed Control of Dynamically Networked Autonomous Devices," IEEE Transactions on Parallel and Distributed Systems, 16 February 2004.

Ramakrishnan, R. and Gehrke, J., *Database Management Systems*, Third Edition, McGraw Hill, 2003.

Rew, R., Davis, G., Emmerson, S. and Davies, H., *The Network Common Data Form Users' Guide, Data Model, Programming Interface, and Format for Self-Describing, Portable Data, NetCDF Version 3.6.1*, Unidata Program Center, May 2005.

Ricard, M. and Kolitz, S., "The ADEPT Framework for Intelligent Autonomy," Intelligent Systems for Aeronautics Workshop, Brussels, Belgium, May 2002.

Rosenblatt, J., "The Distributed Architecture for Mobile Navigation," *Journal of Experimental and Theoretical Artificial Intelligence*, Volume 9, Number 2/3, April-September, 1997.

Russell, S. and Norvig, P., *Artificial Intelligence, a Modern Approach*, Second Edition, Prentice Hall, 2003.

Serin, E., *Design and Test of the Cross-Format Schema Protocol (XFSP) for Networked Virtual Environments*, Master's Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, March 2003. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/theses. Accessed August 2006.

Simmons, R., "NSCT ONE and EOD Small UUV Acquisition Programs," ONR Joint Review of Technology Applicable to Mine Countermeasures and Associated Missions, Panama City, Florida, February 2004.

Spyns, P., Meersman, R., and Jarrar, M., "Data Modeling Versus Ontology Engineering," *Association for Computing Machinery (ACM) Special Interest Group Management of Data (SIGMOD) Record, Special Section on Semantic Web and Data Management*, Volume 31, number 4, December 2002.

Stentz, T., "Carnegie Mellon University Technology for Mission-Reconfigurable Unmanned Undersea Vehicle (MRUUV), Current project status brief at the Naval Undersea Warfare Center," Newport, RI, 28 July 2004.

Stevens, B. L. and Lewis F. L., *Aircraft Control and Simulation*, Second Edition, John Wiley and Sons, 2003.

Stokey, R. P., *Interoperable Command Set for Multiple Vehicles (Compact Control Language)*, ONR Joint Review of Technology Applicable to Mine Countermeasures and Associated Missions, Panama City, Florida, February 2004.

Stokey, R. P., "A Compact Control Language for Autonomous Underwater Vehicles," Woods Hole Oceanographic Institution, April 2005. Available at http://acomms.whoi.edu/40x%20Specifications/401100%20Compact%20Control%20Language/CCL%20April%202005%20Public%20Release%201.0.pdf. Accessed August 2006.

Stokey, R. P., Freitag, L. E. and Grund, M. E., "A Compact Control Language for AUV Acoustic Communication," *Proceedings of the Oceans '05, Europe Conference*, June 2005. Available at http://acomms.whoi.edu/40x%20Specifications/401100%20Compact%20Control%20Language/CCL_OceansEurope05-Final.pdf. Accessed August 2006.

St. Peter, M. and LaPointe K. M., "Swarming Mobile Searchers," 72nd Military Operations Research Society Symposium, Unmanned Systems Working Group Presentation, Monterey, California, June 2004.

Sun Microsystems, Inc., *Java Architecture for XML Binding (JAXB) Version 1.0 Online Users Guide*, 2005. Available at http://java.sun.com/webservices/jaxb/users-guide/jaxb-using.html. Accessed August 2006.

Tidwell, D., *XSLT, Mastering XML Transformations*, O'Reilly and Associates, 2001.

Turner, E. H. and Chappell, S. G., "Conceptual Communications for Multi-Vehicle Systems," University of New Hampshire (UNH) Technical Report #95-08, May 1995.

Turner, R. M., Blidberg, D. R., Chappell, S. G. and Jalbert, J. C., "Generic Behaviors: an Approach to Modularity in Intelligent Systems Control," *Proceedings of the 8th International Symposium on Unmanned and Untethered Submersible Technology*, Durham, NH, 1993.

Turner, R. M. and Turner, E. H., "Self-Organization and Reorganization of Multi-AUV Systems: CoDA Project Overview," *Proceedings of the 2004 IEEE/OES Autonomous Underwater Vehicles Symposium*, Sebasco, Maine, June 2004.

Vaughan, R., Gerkey, B., and Howard, A., "On Device Abstractions for Portable, Reusable Robot Code," *Proceedings of the 2003 IEEE/RJS International Conference on Intelligent Robots and Systems*, Las Vegas, Nevada, October 2003.

Weekley, J., Brutzman, D., Healey, A., Davis, D., and Lee, D., "AUV Workbench: Integrated 3D for Interoperable Mission Rehearsal, Reality and Replay," *Proceedings of the Mine Warfare Association Australian-American Mine Warfare Conference*, Canberra, Australia, February 2004. Available at http://xmsf.cvs.sourceforge.net/xmsf/AuvWorkbench/documentation/papers. Accessed August 2006.

Werger, B., "Ayllu: Distributed Port-Arbitrated Behavior-Based Control," *Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems*, Knoxville, Tennessee, October 2000.

World Wide Web Consortium, XSL Transformations (XSLT) Version 1.0 Recommended Specification, edited by Clark, J., October 2001. Available at http://www.w3.org/TR/2001/REC-xsl-20011015. Accessed August 2006.

World Wide Web Consortium (W3C), Extensible Markup Language (XML) 1.0 (Third Edition) Recommended Specification, edited by Bray, T., Paoli, J., Sperberg-McQueen C. M., Maler, E., and Yergeau, F., February 2004. Available at http://www.w3.org/TR/REC-xml. Accessed August 2006.

World Wide Web Consortium, XML Schema Part 1: Structures Second Edition Recommended Specification, edited by Biron, Thompson, H. S., Beech, D., Maloney, M. and Mendelsohn, N., October 2004. Available at http://www.w3.org/TR/xmlschema-1. Accessed August 2006.

World Wide Web Consortium, XML Schema Part 2: Datatypes Second Edition Recommended Specification, edited by Biron, P. V. and Malhotra, A., October 2004. Available at http://www.w3.org/TR/xmlschema-2. Accessed August 2006.

World Wide Web Consortium, Document Object Model Level 3 Core Specification Recommendation, edited by Le Hors, A., Le Hegaret, P., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S., April 2004. Available at http://www.w3.org/DOM. Accessed August 2006.

World Wide Web Consortium, OWL Web Ontology Language Overview Recommendation, edited by McGuinness, D. L. and van Harmelen, F., February 2004. Available at http://www.w3.org/TR/owl-features. Accessed August 2006.

World Wide Web Consortium, XML Binary Characterization, edited by Goldman, O. and Lenkov, D., March 2005. Available at http://www.w3.org/TR/xbc-characterization. Accessed August 2006.

Yang, L., Kreamer, W., Adams, M., Carr, F., Guerra, C., Page, L., McConley, M., Hall, W. and Falcone, C., "Hierarchical Planning for Large Numbers of Unmanned Vehicles," *The Draper Technology Digest*, Volume 9, Draper Laboratory, 2005.

Yumetech, Inc., "Xj3D Picking Extensions," Proposed extension to the X3D specification, 2004. Available at http://www.xj3d.org/extensions/picking.html. Accessed August 2006.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Associate Professor Don Brutzman
   Department of Undersea Warfare
   Naval Postgraduate School
   Monterey, CA

4. Professor Neil Rowe
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

5. Professor Robert McGhee
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

6. Associate Professor Christian Darken
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

7. Distinguished Professor Anthony Healey
   Department of Mechanical Engineering
   Naval Postgraduate School
   Monterey, CA

8. Assistant Professor Kevin Squire
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

9. Dr. Kwang Song
   Department of Mechanical Engineering
   Naval Postgraduate School
   Monterey, CA

10. Mr. Douglas Horner
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA

11. Mr. Sean Krageland
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA

12. Mr. Curt Blais
MOVES Institute
Naval Postgraduate School
Monterey, CA

13. CAPT Dan Gahagan, USN
Naval Research Laboratory
Arlington, VA

14. CAPT Dennis Sorensen, USN
Office of Naval Research
Arlington, VA

15. Dr. Thomas Swean
Office of Naval Research
Arlington, VA

16. Dr. Thomas Curtin
Office of Naval Research
Arlington, VA

17. Mr. John Moore
Navy Modeling and Simulation Office
Arlington, VA

18. Mr. D. Richard Blidberg
Autonomous Undersea Systems Institute
Durham, NH

19. Mr. Steven Chappell
Autonomous Undersea Systems Institute
Durhman, NH

20. Mr. Rick Komerska
Autonomous Undersea Systems Institute
Durhman, NH

21. Mr. Peter Flynn
Naval Research Laboratory,
Stennis Space Center, MS